



EXPLORING HARDWARE-BASED PRIMITIVES
TO ENHANCE PARALLEL SECURITY MONITORING
IN A NOVEL COMPUTING ARCHITECTURE

THESIS

Stephen Mott, Second Lieutenant, USAF

AFIT/GE/ENG/07-17

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GE/ENG/07-17

EXPLORING HARDWARE-BASED PRIMITIVES TO ENHANCE
PARALLEL SECURITY MONITORING IN A NOVEL COMPUTING
ARCHITECTURE

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
In Partial Fulfillment of the Requirements for the
Degree of Master of Science in Electrical Engineering

Stephen Mott, B.S.E.E.
Second Lieutenant, USAF

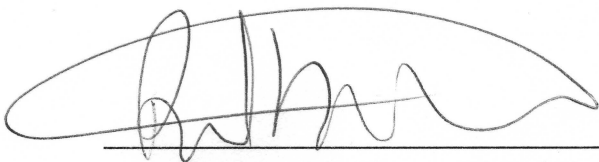
March 2007

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

EXPLORING HARDWARE-BASED PRIMITIVES TO ENHANCE
PARALLEL SECURITY MONITORING IN A NOVEL COMPUTING
ARCHITECTURE

Stephen Mott, B.S.E.E.
Second Lieutenant, USAF

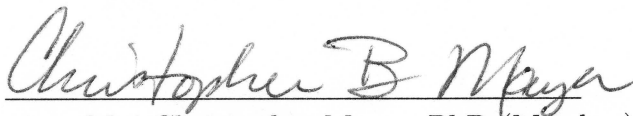
Approved:



Maj Paul Williams, PhD (Chairman)

5 MAR 07

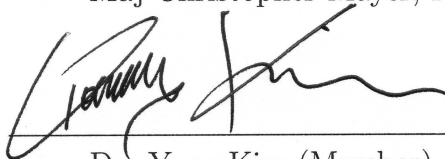
date



Maj Christopher Mayer, PhD (Member)

5 MAR 07

date



Dr. Yong Kim (Member)

5 MAR 07

date

Abstract

Maintaining computer security is an ever present problem in today's increasingly connected world. As computer architectures increase in complexity to support the needs of ever more complex applications, it is becoming more difficult to protect against misuse and attack. Software-based security monitoring mechanisms have been implemented, however, these are circumventable, have slow time-to-detect, and degrade performance of the system being monitored. To overcome these shortcomings, our research focuses on moving security-related monitoring mechanisms from software to hardware.

This research explores how hardware-based primitives can be implemented to perform security-related monitoring in real-time, offer better security, and increase performance compared to software-based approaches. In doing this, we propose a novel computing architecture, derived from a contemporary shared memory architecture, that facilitates efficient security-related monitoring in real-time, while keeping the monitoring hardware itself safe from attack. This architecture is flexible, allowing security to be tailored based on the needs of the system. We have developed a number of hardware-based primitives that fit into this architecture to provide a wide array of monitoring capabilities. A number of these primitives provide capabilities, such as multi-context monitoring and virtual memory introspection, that were not previously possible at the hardware level. Not only does this allow for more robust security-

related monitoring when compared to software-based approaches, it also allows the security-related monitoring concepts presented in this research to be applied across a broad range of computing environments.

A number of these primitives are implemented in the context of our architecture. Experimentation with these prototypes validated our approach and demonstrated real-time performance. However, due to the limitations of current computer architectures, a number of the primitives could not be implemented. In these cases, we describe what is needed for these concepts to be implemented and argue why these primitives will function correctly. Therefore, this research shows that security-related monitoring tasks can be moved from software to hardware in a way that security, system performance/usability, and time-to-detect are all improved compared to software-based methods.

Table of Contents

	Page
Abstract	iv
List of Figures	x
List of Abbreviations	xi
I. Introduction	1
1.1 Background and Problem Overview	1
1.2 Research Goals	4
1.3 Contributions	5
1.4 Document Layout	8
II. Related Research Exploration	10
2.1 Classes of Attack	10
2.1.1 Viruses, Worms, and Trojans	10
2.1.2 Rootkits	11
2.1.3 Timing-based Attacks	12
2.1.4 Relocation Attacks	12
2.2 Intrusion Detection Taxonomies & Categorizations	12
2.2.1 Intrusion Detection Systems: A Survey and Taxonomy	13
2.2.2 A Categorization of CSM Systems and The Impact on The Design of Audit Sources	17
2.2.3 Monitoring System Security	22
2.3 Uniprocessor-based Host Intrusion Detection	25
2.4 Network-based Intrusion Detection	27
2.5 Virtual Machine Monitor-based Intrusion Detection	29
2.6 Coprocessor-based Intrusion Detection	30
2.6.1 Cryptographic Coprocessors	32
2.6.2 Loosely Coupled Monitoring Coprocessors	34
2.6.3 Tightly Coupled Monitoring Coprocessors	38
2.7 Hardware-assisted Security Mechanisms	43
2.7.1 Hardware-based Stack Protection	44
2.7.2 Microinstruction-based Monitoring	45
2.7.3 Control Flow Monitoring	47
2.7.4 Non-executable Memory	51
2.8 Trusted Operating Systems	52

	Page
III. Research Concepts and Methodology	54
3.1 Research Hypothesis	54
3.2 Targeted Exploits	55
3.3 Architectural Overview	55
3.3.1 Hardware Architecture	55
3.3.2 Shadow Monitoring Unit Configuration	58
3.3.3 Software Architecture	59
3.3.4 Monitoring System Security	61
3.4 Target Environment	63
3.5 Functional Primitives	63
3.5.1 Multi-context Hardware Monitors	64
3.5.2 Program Counter and Instruction Trace Exposure	66
3.5.3 Peripheral Access Control	68
3.5.4 Hardware-based Memory Introspection	69
IV. Research Platform and Implementation	80
4.1 Purpose of Implementation	80
4.2 Development Platform	81
4.2.1 Embedded Processors	81
4.2.2 CoreConnect Bus Architecture	85
4.2.3 Software Support	88
4.3 Linux Implementation	90
4.3.1 Embedded Linux 2.4	91
4.3.2 uCLinux	91
4.4 Functional Primitives	92
4.4.1 Execution Policy Enforcement	93
4.4.2 Multi-context Hardware Monitors	95
4.4.3 Peripheral Access Control	99
4.4.4 Asymmetrically Shared Main Memory	102
4.4.5 Co-opted Memory Management Unit	108
4.4.6 SMU with Multiple MMUs	111
V. Testing and Results	116
5.1 Execution Policy Enforcement Module	116
5.1.1 Testing Methodology	116
5.1.2 Simulation	117
5.1.3 Implementation Results	119
5.2 Multi-context Hardware Monitors	121
5.2.1 Testing Methodology	121
5.2.2 Simulation	124

	Page
5.2.3 Implementation Results	125
5.3 Peripheral Access Control	126
5.3.1 Expected Results	126
5.4 Asymmetrically Shared Main Memory	127
5.4.1 Testing Methodology	127
5.4.2 Implementation Results	129
5.5 Memory Management Unit Co-opting	130
5.5.1 Benefits and Limitations	130
5.6 Multiple Memory Management Units	132
5.6.1 Benefits and Limitations	132
VI. Conclusion	135
6.1 Conclusions	135
6.1.1 Improved Time-to-Detect	135
6.1.2 Hardness of the Monitoring System	136
6.1.3 Displaced Security Workload	136
6.1.4 Novel Hardware-based Monitoring Techniques	137
6.1.5 Monitoring System Flexibility	138
6.1.6 Monitoring System Extensibility	138
6.1.7 Improved Range of Monitoring Granularity	139
6.2 Future Work	139
6.2.1 Virtual Memory Introspection Implementation	139
6.2.2 Enhanced Debug Registers	140
6.2.3 Forensics Capabilities	141
6.2.4 Automatic Process Repair	142
6.2.5 Minimum Required Resource Investigation	142
6.2.6 Scalability: Multiple PPU's per SMU	143
6.2.7 Security Logic Units	144
Appendix A. Implementation Code	145
A.1 Execution Policy Enforcement Module	145
A.1.1 MB_Trace4mdm_top.vhd	145
A.1.2 noex_mem_en.vhd	148
A.2 Multi-context Hardware Monitoring	149
A.2.1 MB_PID2_top.vhd	149
A.2.2 PID_LOGIC.vhd	155

	Page
Appendix B. Development Software Tutorials	158
B.1 Embedded Linux Tutorial	158
B.1.1 Initializing The Environment	158
B.1.2 The Base System Builder	159
B.1.3 Platform Studio	162
B.1.4 Compact Flash and SysACE	164
B.2 Troubleshooting	166
B.2.1 Development Environment	166
B.2.2 General Linux	167
B.2.3 Embedded Linux Installation	168
Bibliography	171

List of Figures

Figure		Page
3.1.	General Overview of the Hardware Architecture	56
3.2.	General Software Architecture	60
3.3.	Multicontext Monitoring With Multiple Monitors	65
3.4.	Program Counter Monitor High-level Architecture	67
3.5.	Peripheral Access Control Monitor General Architecture	69
3.6.	Memory Map and Permissions as Viewed by the PPU and the SMU	71
3.7.	Co-opted Memory Management Unit High-level Architecture .	75
3.8.	Multi-MMU SMU High-level Architecture	77
4.1.	Example Embedded System Using Core Connect Bus Architecture	86
4.2.	Program Counter Monitor Implementation	94
4.3.	Logical Implementation of PID Retrieval	96
4.4.	An Example of Monitoring Multiple Processes With Multiple Monitors	99
4.5.	Planned Peripheral Access Control Implementation	101
4.6.	PAT and PMT Example	102
4.7.	MPMC2 Basic Organization	103
4.8.	Memory Map and Permissions as Viewed by the PPU and the SMU	105
4.9.	Asymmetrically Shared Memory Implementation	106
4.10.	Proposed Co-opted Memory Management Unit Architecture . .	108
5.1.	Execution Policy Enforcement Simulation Result	118
5.2.	Multicontext Monitors Simulation Result	124

List of Abbreviations

Abbreviation		Page
SPCM	Security Policy Compliance Monitoring	8
ID	Intrusion Detection	10
IDS	Intrusion Detection System	10
CSM	Computer Security Monitor	17
CuPIDS	CoProcessor Intrusion Detection System	21
StUPIDS	Standard UniProcessor Intrusion Detection System	21
OS	Operating System	26
FPGA	Field Programmable Gate Array	28
VMM	Virtual Machine Monitor	29
VMBR	Virtual Machine-based Rootkit	30
PCI	Peripheral Component Interconnect	31
TPM	Trusted Platform Module	33
PCI	Peripheral Component Interface	36
IPC	Instructions Per Clock	37
UMA	Uniform Memory Access	38
SPCM	Security Policy Compliance Monitoring	38
SMP	Symmetric Multiple Processing	38
CPP	CuPIDS Production Process	39
CSP	CuPIDS Shadow Process	39
SECM	Security Enhanced Chip Multiprocessor	41
L2	Level 2	41
L1	Level 1	41
LIFO	Last-In, First-Out	44
ISA	Instruction Set Architecture	45
FSM	Finite State Machine	50

Abbreviation		Page
NX	No-eXecute	52
XD	eXecute Disable	52
PTE	Page Table Entry	52
QA	Quality Assurance	55
PPU	Production Processor Unit	55
SMU	Shadow Monitor Unit	55
RTL	Real-Time-Logic	58
NUMA	Non-Uniform Memory Access	71
MMU	Memory Management Unit	74
PPC405	PowerPC 405fx	81
RISC	Reduced Instruction Set Computer	82
FSL	Fast Simplex Link	83
CCBA	CoreConnect Bus Architecture	85
PLB	Processor Local Bus	85
OPB	On-chip Peripheral Bus	85
DCR	Device Control Register	85
BSP	Board Support Package	87
JTAG	Joint Test Action Group	89
MDM	Microblaze Debug Module	90
BRAM	Block Random Access Memory	93
LMB	Local Memory Bus	93
CPR	Current PID Register	98
PAT	Peripheral Access Table	100
PMT	Peripheral Map Table	100
PIM	Port Interface Module	103
XCL	Xilinx Cache Link	103
CDMAC	Communication Direct Memory Access Controller	104
NPI	Native Port Interface	104

Abbreviation		Page
CR3	Control Register 3	111
SLU	Security Logical Unit	144

EXPLORING HARDWARE-BASED PRIMITIVES TO ENHANCE PARALLEL SECURITY MONITORING IN A NOVEL COMPUTING ARCHITECTURE

I. Introduction

Adding certain functional primitives to current computer architectures will leverage previously unavailable knowledge of system state at the hardware level. This allows for increased computer security with less performance overhead than previously proposed methods, while still maintaining strong security for the security mechanisms themselves. In this paper, we discuss our research that has allowed us to make such claims.

1.1 Background and Problem Overview

Computer security is an ever present problem in today's connected world. Every year the reported instances of vulnerabilities in software grows and there seems to be no end in sight [10]. The computing industry is aware of this and tries to implement good programming practices, create safer programming constructs, as well as modify how operating systems interact with the processes they coordinate. However, programmers are only human and are bound to make mistakes, no matter how strong their resolve is to create non-exploitable code. Furthermore, as code becomes more complex to solve increasingly complicated problems, vulnerabilities are more difficult to prevent. This is in large part due to abilities afforded by the current computer

architecture paradigm - the primary design goal of which has been to improve performance, not security. As such, hardware in recent years has begun to implement changes to contemporary processor architectures that help to prevent certain security exploits such as buffer overflows. However, current hardware-based security techniques implemented in commercial processors are far from exhaustive solutions. As a result, we need more robust security in today's computing systems, but increased security introduces a number of problems.

Adding security to a system does not come for free. Increased security typically increases cost in terms of performance as well as usability. Moreover, there is usually a distinct inverse relationship between performance/usability and the level of security a system can provide. That is, a system with a high level of security does so at the expense of lower performance, and vice versa. This is due to the sharing of resources between security-related and non-security-related tasks. Thus a shift towards using dedicated hardware to monitor another processor for security purposes is needed.

A large amount of research has already been conducted on security mechanisms that utilize dedicated hardware - much of which is described in Chapter II. These mechanisms all leverage hardware to process state information, but may differ in how such state information is gathered. Some hardware monitors gather state information in software, whereas others gather system state at the hardware level. Each way of gathering state has its advantages and disadvantages which result from the level of the abstraction that the gathered state information corresponds to. State information at the hardware level is seen only as instructions, data, and control signals. As such,

state gathered at the hardware level corresponds to the lowest level of abstraction. State information gathered at the software level corresponds to a higher abstraction level. It is this higher level of abstraction that allows software-based techniques to better correlate the gathered state to what the monitored code is actually doing, putting the monitored state into context easier. Context can be and is determined for state information gathered at the hardware level as well, however, it is more difficult to do so than using software-based methods.

Hardware-based security monitors that gather state at the hardware level can gather state in real-time (as it is implemented in hardware), but due to the lack of abstraction at this level, it is difficult to determine the context of the state information gathered (i.e., what the state actually means in relation to the system). This typically limits the kinds of monitoring that these mechanisms can perform as well as limits the environments they are normally applied to. Hardware-based security monitors that gather state information at the software level retrieve state information that can inherently contain more contextual information. As a result, such mechanisms can be applied in more complex computing environments. However, this higher abstraction level (i.e., greater context) is gained by the monitoring software being tightly coupled to the code it is monitoring. This tends to decrease performance of the monitored system as well as decrease the security of the monitoring system/mechanism itself, however.

1.2 *Research Goals*

Our research specifically targets the aforementioned issues by exploring ways in which we can alter the currently accepted computer architecture model in an effort to increase computer security. This is accomplished by breaking through the limitations that current computing architectures impose by providing new methods by which useful system state information can be revealed and processed in parallel, enabling real-time security-related monitoring. We have developed a novel computing architecture derived from a contemporary shared memory multiprocessor model that provides for the implementation of a number of functional primitives in hardware that we leverage to be able to provide such capabilities while maintaining compatibility with the current computing model.

To help facilitate better overall system security, we intend to increase the security of the monitoring hardware itself. By protecting the monitoring hardware, we can ensure the correct operation of the monitoring hardware to a greater degree than software-based monitoring methods. We plan to accomplish this by tightly coupling the monitoring hardware to the hardware executing the monitored code in order to gather context-rich state information, rather than coupling the monitoring software to the software being monitored. This allows the monitoring system to remain as transparent to the monitored system as possible. Consequently, we minimize the attack surface of the monitoring hardware itself, reducing the chance that the monitoring system can be compromised [38].

Monitoring with dedicated hardware allows code to be monitored parallel as it executes on the monitored processor, which should enable a real-time security monitoring capability. Additionally, as we intend to keep software coupling to an absolute minimum, we believe little to no added overhead will be imposed on the system compared to systems that couple the monitoring software to the monitored code more tightly. As a result, no more than a minimal impact on the system’s usability would likely occur.

1.3 Contributions

In working towards our research goals, we make a number of contributions. They are as follows:

- Developed a novel, security-oriented computing architecture which is flexible, secure, and extensible. The architecture is specifically designed to allow context-rich state information to be gathered, while keeping the monitoring hardware as secure as possible.
- Created a categorization of monitoring system security. This benefitted our research when designing our architecture to provide the best balance of security and capabilities for the monitoring hardware.
- Designed a number of functional primitives that fit into the architecture. All primitives are based in hardware and can provide monitoring in real-time. It should also be mentioned that many of the primitives are complementary to each other and can be implemented together in varying combinations. Thus,

security can be tailored to a particular application's needs. The primitives are briefly described below.

Multi-context Hardware Monitors: This primitive allows the monitoring hardware to be able to discern between different processes executing on the monitored processor - a capability previously not possible at the hardware level. As a result, hardware-based monitoring mechanisms can be implemented in a broader range of computing environments.

Execution Policy Enforcement Module: This primitive prevents malicious code from executing. Although some computing architectures contain this capability, this primitive can add such a capability to processors that do not natively support it.

Peripheral Access Control: This primitive ensures that processes do not access system devices that they were not originally intended to access.

Asymmetrically Partitioned Main Memory: This primitive allows memory to be shared in an asymmetric manner. This provides the monitoring system with visibility into the physical memory space of the monitored processor, while preventing the monitored processor from having visibility into the monitoring system's memory space.

MMU Co-opting: This primitive provides the monitoring hardware with visibility into the virtual memory space of the process that is currently executing on the monitored processor - a capability previously not possible

at the hardware-level. As a result, certain forms of monitoring, such as invariant checking, can be performed on both user-level and kernel-level processes.

Monitoring Using Multiple MMUs: This primitive enables the same capabilities as MMU co-opting, however, it can also provide visibility into the virtual memory space of processes not currently executing. This provides for a number of novel security-related capabilities, such as trusted process execution (throughout the process' entire runtime) and real-time deadlock detection, among others.

- A number of the primitives were implemented to show proof of concept.

While the aforementioned contributions are physical results of our research, we also make a number of contributions to the security-related monitoring field in general. They are as follows:

Time-to-Detect: Our primitives can provide for real-time security monitoring. As a result, the primitives can provide improved time-to-detect compared to software-based methods.

Hardness of Monitor in the Presence of Malicious Code: The security of the monitoring system itself can be ensured, to a good degree, in the event that the monitored system has been compromised by malicious code. As a result, the monitoring system can continue to function in such a case.

Types of Inputs to the Monitoring System: We devised new ways to gather state information at the hardware level. This increases the types of inputs (at the hardware level) to the monitoring system over previous methods, resulting in more robust security-related monitoring capabilities.

Range of Monitoring Granularity: Our primitives can allow monitoring granularity ranging from the individual instruction level to the process level. As a result, this research increases the range of monitoring granularity that can be provided via hardware-based mechanisms. This allows our primitives to provide security policy compliance monitoring (SPCM) in a broad range of computing environments.

1.4 Document Layout

This chapter covered the general area of our research, what problems we are trying to solve, and why it is important to do so. Additionally, we outline our research goals and detail our contributions. Chapter II describes work done by other researchers in the same or related fields that we used as a basis in forming our own work. We present the actual thesis statement, research methodologies, and theories behind the implementation in Chapter III. Chapter IV covers in detail the actual implementation of our work, while Chapter V presents the testing methodology, any applicable simulations, the results, and analysis of the implementation. The concluding remarks as well as future work areas our research has opened are presented in Chapter VI. Appendices are also included at the end of the document and contain

information not appropriate for the main document such as code created through our research efforts and tutorials for our development environment.

II. Related Research Exploration

This chapter presents the results of research that we found useful in conducting our own exploration. We begin by presenting work that classifies and describes the different forms of intrusion detection (ID). These concepts are then expanded by presenting the different ways intrusion detection systems (IDS) have been implemented. The section concludes with descriptions of the various hardware-based security mechanisms that have been proposed thus far.

2.1 *Classes of Attack*

In order to help describe the various monitoring mechanisms we propose in this document, it is useful to understand some of the various forms of malicious attacks and the vehicles employed. While there are countless forms of attack, we attempt to summarize the different classes of attacks in this section. This is not meant to be an exhaustive list, but rather a number of attacks that are useful when describing our work that can be referenced when needed.

2.1.1 Viruses, Worms, and Trojans. CACI International provides a breakdown of various types of computer security threats in [9]. [9] defines viruses as a form of malicious software that attaches itself to other software within a system. Viruses are not self-propagating across machines, and thus have no means by which to spread to another system unless copied to another location by some means external to the virus itself. Worms are standalone programs that perform some malicious function within a system. Worms, unlike viruses, have the ability to propagate themselves to

other systems via a network. Trojans are malicious software masquerading as useful software. Trojans can be implemented as a worm (self propagating) or as a virus that is attached to a piece of software prior to distribution. While viruses, worms, and trojans are all slightly different, they are all related in that they actively execute code in an unintended fashion. They act as the basic tools to perform malicious activities within a system.

2.1.2 Rootkits. The formal definition of a rootkit reads, “A rootkit is a set of software tools intended to conceal running processes, files or system data from the operating system” [57]. Although rootkits can be used for non-malicious purposes, we are only concerned with the malicious use of rootkits. Rootkits are typically used by attackers to keep “root” access to a computer system - the highest privilege level - and hide their activities in order to prevent detection by a system administrator. Rootkits are typically installed onto a computer via a security vulnerability and are very noticeable the first time the attacker gains root level access. Once installed, the rootkit cleans evidence of its initial entry and provides an exploit (i.e., a backdoor in many cases) to the system using common commands that have been modified into trojans. These backdoors allow the attacker to continue to access the system without being noticed [7]. As rootkits can provide a means for an attacker to have complete control over an entire computer system while making detection difficult, rootkits have become regarded as highly dangerous, making rootkit defense a very active area of research.

2.1.3 Timing-based Attacks. Timing-based attacks are any type of malicious attack whose operation is intended to focus on a vulnerability associated with the timing of particular events within a system. For example, some detection systems only check for malicious activity at specific time intervals. It is possible for an attacker to target the window of time where the code is not being monitored. Moreover, such a vulnerability makes it possible for any malicious software used to be removed and all traces of illegitimate activity to be erased before the monitor is ever invoked [46]. However, for such an attack to occur, the attacker/malicious software must know when a particular system is vulnerable at a certain point in time in order to exploit that vulnerability.

2.1.4 Relocation Attacks. A relocation attack relocates the malicious code to avoid detection. Relocation is done typically to somewhere that cannot be monitored. For example, malicious code may be detected in memory, so the malicious code is engineered in such a way that it executes from within a processor's cache [46]. This type of attack seems particularly difficult to implement, but it is a possible threat nonetheless.

2.2 Intrusion Detection Taxonomies & Categorizations

While there are many varying definitions of intrusion detection, we consider intrusion detection to be the identification of abnormal system behavior given an idea of what good and/or bad system behavior should be. Despite this singular definition, there are numerous forms that intrusion detection can take, and a variety of systems

which operate differently and to different degrees. In order to help others understand the key differences between these different intrusion detection systems, as well as the general concepts of intrusion detection and security policy compliance monitoring, work has been done to classify the features of such systems proposed thus far. The results pertinent to our research are described below.

2.2.1 Intrusion Detection Systems: A Survey and Taxonomy. In [4], Axelsson provides a comprehensive breakdown of intrusion detection principles which he uses to survey and classify numerous intrusion detection systems that have progressed to the prototype stage of development. He asserts that in order to develop methods to detect intrusions, one must first know what to look for. This is not as easy a problem as one might first think, as Axelsson attests to. He points out that not only are some threats unknown (and hence unforeseeable), but also that even known threats can be difficult to distinguish from what is considered normal operation. Furthermore, it is never entirely certain what the source of an attack will be, whether it is an assailant hacking into a network, a user within the network that is abusing privileges, etc. Perhaps one of the largest problems when implementing an IDS is the lack of useful information provided to the IDS.

Knowing the problems associated with designing intrusion detection systems, Axelsson produced a taxonomy of intrusion detection by generally characterizing intrusion detection principles into two main classes: 1) anomaly-based detection and

2) signature-based detection. This taxonomy, however, can be extended to include a principle known as specification-based detection, first described by Ko in [31].

- Anomaly-based Detection: According to Axelsson, anomaly-based detection does not look at the the actual intrusion itself, but rather the reaction of the process in question to an intrusion [4]. It operates on the assumption that an intrusion will produce abnormal behavior within a system, and that the abnormal behavior can be considered suspicious. Thus, an anomaly-based intrusion detector must know what constitutes abnormal behavior, as well as at what point to deem abnormal behavior as an intrusion.

To determine what is considered normal behavior, Axelsson breaks down anomaly detection into two types: 1) self-learning and 2) programmed. In the first type of anomaly detection, the process under scrutiny is run in a safe environment for an extended period of time. As the process executes, the IDS gathers statistics on that process' operation in order to build a model of normal operation for that process. The system is then placed into use and monitored, signaling a violation when an event outside of the previously gathered behavior data occurs. The second type of anomaly detection depends on a system administrator, designer, and/or user to teach the system specifically what constitutes abnormal behavior and how to signal a security violation. Thus the user, rather than the system itself in the self-learning case, determines what constitutes abnormal behavior.

- **Signature-based Detection:** Signature-based detection (also known as misuse detection) relies on the user to provide a model of an intrusive event to the intrusion detection system. A signature-based detector will look for known specific clues left behind from an intrusive process in order to determine if an intrusion has occurred. With that said, signature-based detectors detect intrusions irrespective of what the normal behavior for the system is. Thus, even normal behavior can flag a security violation if such behavior matches a provided model of illicit activity. As such, the models used for a signature-based detector must be very precise so as to ensure low false positive rates.

Since signature-based detectors rely on models of known threats, intrusion detection systems using this principle can only be programmed to know what to look for. This can be done via state modeling, expert-system, string matching, or a simple rule-based method. State modeling consists of a number of states occurring within a system which indicates whether an intrusion has taken place. An expert-system reasons about the security state of the system given rules that describe intrusive behavior. String matching is an inflexible, yet simple means to detect intrusions via comparing substring text received by the system. The simple rule-based approach is a less complex version of an expert-system that often leads to a faster execution.

- **Specification-based:** Specification-based detection attempts to merge the high detection rate of signature-based detection with the ability to detect novel attacks of anomaly-based detection [20]. In systems with explicitly defined security

policies, specification-based detection can be used to detect any deviations from that security policy. At the time Axelsson's taxonomy was created, specification-based detection had not been widely accepted. As noted by Williams in [59], this is due to a lack of clearly defined security policies at the time, although more recently that is beginning to change.

Axelsson points out that most intrusion detection systems studied fall into more than one category. He claims this is not due to his taxonomy being vague, but rather that many intrusion detection systems created thus far employ multiple intrusion detection principles. This taxonomy also makes evident two orthogonal concepts in intrusion detection: 1) anomaly versus signature and 2) programmed versus self-learning. As Williams notes in [59], with the inclusion of specification-based detection, the first concept must be modified to anomaly versus signature versus specification.

Axelsson then goes on to classify intrusion detection systems by what type of intrusion they most readily detect. From this, three forms of intrusion are derived:

1. Well-known Intrusions: Intrusions that exhibit a static and well defined pattern.

These generally take little work to detect.

2. Generalizable Intrusions: Intrusions that allow for some degree of variability in how they are executed. These often exploit a general flaw or set of flaws in a process rather than a specific vulnerability.

3. Unknown Intrusions: Intrusions that have a very weak coupling to a specific flaw or an extremely general flaw. Thus, the intrusion detection system does not necessarily know what to expect.

Axlessen ends with a taxonomy of system characteristics. This is very similar to some characteristics described in Kuperman's Ph.D. work in [33]. One characteristic that is of importance to this research, but not present in Kuperman's work (See Subsection 2.2.2 for the other characteristics) is processing granularity. The processing granularity of an intrusion detection system describes how much and/or how fast data is processed by the intrusion detection system. The two main categories are batch granularity and continuous granularity. Batch granularity processes data in chunks. This helps to decrease overhead of the intrusion detection system, but can add to the time to detect. Conversely, continuous granularity processes all data as it is produced. This can impose a large overhead to the system being monitored, but generally has better time-to-detect compared to a similarly configured intrusion detection system using batch processing granularity. Since this affects the effectiveness and speed of detection, careful attention must be paid to this characteristic in our research.

2.2.2 A Categorization of CSM Systems and The Impact on The Design of Audit Sources. Kuperman presents a characterization of computer security monitors (CSM) in [33]. His work includes characteristics of such computer security monitors which are of importance to our work. Two characteristics - the goal of detection and the timeliness of detection - were found to be of importance to Williams' previous

work in [59]. As our research is based heavily on Williams' research, these characteristics are also of importance to our research and are described below. Furthermore, for our work we add a new characteristic to these - Monitoring System Security - and is also described below.

2.2.2.1 Goal of Detection. In order to categorize security monitoring systems, Kuperman first asks the question *For what security purpose is this system monitoring behavior?* To answer this question, he identifies a number of major areas of focus within the computer security monitoring field. These areas are described below:

- Detection of Attacks: Detects attempts to exploit a specific vulnerability in a computer system. Attacks can be in the form of a virus, trojan, etc. and are intended to cause harm to the system or use it in an illegitimate manner.
- Detection of Intrusion: Relies on the notion of legitimate users of a specific computer system. Intrusions can be external to the system (e.g., over a network connection) or internal to the system (i.e., from the system itself but by an unauthorized user). This is also known as an intrusion detection system.
- Detection of Misuse: Similar to detection of intrusion, however, the misuse being detected is by an authorized user of the system. Thus, no intrusion is committed nor detectable, but illegitimate actions are detected.
- Computer Forensics: Gathering of data to reconstruct previously occurred activities on a system. This could be used to determine if, when, and how an

attack, intrusion, or misuse occurred, but can also be used for other legitimate actions such as message verification or the tracking of changes made by the computer's administrator.

Our research primarily focuses on the detection of attacks, intrusion, and misuse. Although our work can also apply to computer forensics applications, that is outside the scope of our research for the time being.

2.2.2.2 Timeliness of Detection. Another way Kuperman categorizes CSM systems is by the timeliness of detection. He proposes a view of the overall system as an ordered set of events. Thus, detection times are described in logical time, rather than temporal time. Using Kuperman's notation, the set of all events taking place in a system is denoted as E . The set of suspect events B is a subset such that

$$B \subseteq E \tag{2.1}$$

and there exists events a , b , and c such that

$$a, b, c \in E \tag{2.2}$$

$$b \in B \tag{2.3}$$

The time at which event x occurs is denoted by t_x . The notation $x \longrightarrow y$ denotes that the event y is causally dependent on the event x . Unless otherwise noted, we

assume the dependence of events occurs in alphabetical order as

$$a \longrightarrow b \longrightarrow c \quad (2.4)$$

$$t_a < t_b < t_c \quad (2.5)$$

Furthermore, it should be mentioned that a may not necessarily be the cause of b and so on. Lastly, the detection function, $D(x)$, is used to determine the truth of the statement $x \in B$.

Using the terminology mentioned above, Kuperman describes four main timeliness categories in his CSM categorization. These categories are described below:

- Real-time Detection: Detection of a bad event b takes place while the system is operating and is further restricted to mean that detection of b occurs before any events that are dependent on b take place. As a result, real-time detection requires the ordering

$$t_b < t_{D(b)} < t_c \quad (2.6)$$

- Near Real-time Detection: Detection of b occurs within some finite time δ of the occurrence of b . Thus, near real-time detection requires the ordering

$$|t_b - t_{D(b)}| \leq \delta \quad (2.7)$$

- Periodic: Event records are analyzed by a security system once every time interval p where p is ordinarily on the order of minutes or hours. Furthermore, the detection must take place before the next set of event records is analyzed in order to prevent an increasing backlog of events causing the security system to fail. This results in the ordering

$$t_{D(b)} \leq t_b + 2 * p \quad (2.8)$$

- Retrospective: Detection of bad events takes place outside of any particular time bounds. Analysis operations typically take place using archived events.

The CoProcessor Intrusion Detection System (CuPIDS) architecture that our work is based upon improves the detection rate over a standard uniprocessor intrusion detection system (StUPIDS) for the same detection function $D(x)$ by being able to perform detection within Kuperman’s real-time detection category [59]. This is due to security monitoring occurring in parallel as the monitored code executes. As our functional primitives perform security monitoring in a similarly parallel manner, they can perform detection within Kuperman’s real-time detection category as well. However, our more hardware-centric methods reduce the detection function’s reliance on software-based methods for process-monitor communications. Not only does this guarantee real-time detection, but also provides an improvement in time-to-detect and detection efficiency over CuPIDS.

2.2.3 Monitoring System Security. A monitoring system can be categorized according to how it provides security for a system, however, there has been no categorization for the security of the monitoring system itself. While overlooked, the security of the monitoring system itself is critical, as the entire system can become vulnerable to attack if the monitoring system itself is compromised. As such, we add our own categorization of monitoring system security to what Kuperman has already proposed in [33]. There are eight levels of monitoring system security ranging from least secure to most secure, each of which is described below.

- **Open:** The monitored system has knowledge of and explicitly coordinates and shares state information with the monitor. No security mechanisms are present to protect the monitor from being compromised. This is the worst case. Monitors at this security level tend to be uniprocessor host-based intrusion detection systems which are discussed in Section 2.3.
- **Soft Security:** The monitored system has knowledge of and explicitly coordinates and shares state information with the monitor. The monitor is secured only by software techniques. The monitor can be compromised without having to first compromise the monitored system. As with the open security level, monitors with this security level tend to be uniprocessor host-based intrusion detection systems which are discussed in Section 2.3.
- **Passive Security:** The monitored system is not necessarily aware of the monitor. Any vulnerability to the monitor is by virtue of how it actually performs

monitoring. As such, information about how the monitor actually operates on gathered state data must be known. Most network IDSs can be considered passively secure as they only monitor network traffic, however, the network traffic can contain information that can actually disable the network IDS when it is processed. Such IDSs are discussed in Section 2.4.

- **Self Security:** The monitored system has knowledge of and explicitly coordinates and shares state information with the monitor. By virtue of how the monitor operates, it provides itself with security. Thus, the monitored system must first be compromised before the monitor itself can be compromised. Software-based techniques can also be used to enhance the security of a self secure monitor. CuPIDS presented in [59] and discussed in 2.6.3.1 is one such system at this monitoring system security level.
- **Loose-hard Security:** The monitored system has knowledge of and explicitly coordinates and shares state information with the CSM. Dedicated hardware mechanisms or a combination of hardware and software techniques exist to protect key portions of the CSM from being compromised. Hardware-based return address stacks (presented in [34]) are an example of a type of monitor with this level of security and are discussed further in Section 2.7.1.
- **Semi-hard Security:** The monitored system has very little or no knowledge of the monitor. As such, the monitor can not be executing on the same processor core as the software being monitored and hardware must be used for communications. The monitored system explicitly coordinates with the monitor via

mechanisms like unmaskable interrupts, but is kept to a minimum. The monitored system's state information is implicitly communicated to the monitor. The monitor cannot be compromised via the system being monitored, but it can be worked around if code controlling synchronization signals to the monitoring is altered (i.e., the monitor will not know when or how to monitor). If this occurs, the monitor can still operate, albeit in a diminished capacity. This is the monitoring system security level that our own work specifically targets.

- **Strict-hard Security:** The monitored system has very limited or no knowledge of the monitoring system. As such, the monitor can not be executing on the same processor core as the software being monitored and hardware must be used for communications. The monitor only observes the operation of the system and has to know when and where to gather specific state information. As such, the operation of the monitor has no dependence on the monitored system. Only a system admin can explicitly communicate with the monitor via a dedicated hardware path such as a communications (COMM) port that only the monitor has access to. CoPilot (presented in [46]) and the Independent Auditors (presented in [40]) are two such systems at this CSM security level. They are discussed further in Section 2.6.2.
- **Complete Security:** This is the ideally secure case. The monitoring system has no contact with the outside world, hence it is self defeating as the system would be completely unusable (i.e., an impenetrable lead box).

While each level of monitoring system security is generally considered more secure than the previous, in many cases there tends to be a tradeoff between the security of the monitor and the ease by which state information can be gathered for monitoring purposes. For example, we consider soft security to allow for easier state retrieval since a monitor with soft security tends to closely couple the monitoring software to the software it is monitoring. Semi-hard security can be considered as having more difficulty gathering state information than monitors with soft security because the monitoring software is completely independent of the software it is monitoring. It should be noted that increased difficulty in gathering state information does not necessarily translate into less overall monitoring functionality as monitoring systems with strict-hard security can provide monitoring that soft-secure monitoring systems cannot. However, there becomes a point where the amount of security actually hampers the kinds of monitoring that can be performed. We consider this point to be at the strict-hard security level. As a result, our work specifically targets the semi-hard security level as it provides the monitoring system with the most security while still allowing for some explicit communication - which aids in gathering context-rich state information - used to synchronize the monitor with the monitored system - a critical ability for our parallel monitoring techniques.

2.3 Uniprocessor-based Host Intrusion Detection

Host-based intrusion detection systems were the first form of intrusion detection. Some examples include Haystack, Tripwire, NIDES, Janus, and IDIOT presented in [1,

18, 27, 32, 52], respectively. Host-based intrusion detection systems are characterized by the fact that the intrusion detection system executes on the same hardware as the code that it monitors. In traditional (i.e., earlier) host-based IDSs, the IDS was integrated into the host operating system (OS) or other software being monitored. This close coupling of the IDS and the code that it monitors is the source of both its greatest strength as well as its greatest weakness. As monitoring code executes on the same hardware as the production code, a host-based IDS allows the easiest access into “context-rich” system state and audit data. However, if an intrusion does actually occur, the intrusion detection system itself is made vulnerable to attack due to its integration with the production software. If the IDS is compromised, the result of such an attack may not even be detectable, leaving a false sense of system security. As a result, most host-based intrusion detection systems fit into either the open or soft monitoring system security levels.

Overall host performance as well as IDS efficiency are also affected by the monitor and production code sharing hardware resources. This is due to interleaving execution of the production code and the monitoring code because only one process can be executing on a uniprocessor at any given time. As a result, multiple processes may be scheduled to execute after an intrusion occurs, allowing the malicious code time to damage the system before the monitoring process can execute. Additionally, granularity is reduced in some cases as the previous scenario also allows for certain types of attacks to erase all traces of their existence prior to the monitor executing, rendering the intrusion completely undetectable [59].

More recently, host-based intrusion detection systems have seen a resurgence in popularity. One such effort is the BlueBox system presented by Chari and Chang in [11]. BlueBox is a policy-driven host-based IDS. Rather than specifically monitor an executing process, BlueBox modifies the core OS of the system such that every system call must first be checked with a binary rule file before it can be invoked. This ensures that no illegitimate system calls can be made by a process unless it specifically has been given permission in its execution policy. As such, every process requires its own set of rules and the security policy effectiveness is reliant on how well the policy is defined by the system administrator creating the policy.

2.4 Network-based Intrusion Detection

Network-based intrusion detection systems have also been an active area of research. They are characterized by analyzing network traffic for known attacks. Details of such examples can be found in [6, 12, 13, 22, 35, 53, 58]. Whereas host-based intrusion detection systems attempt to protect only one system, a single network intrusion detection system can protect an entire group of systems from attacks. Network IDSs are usually placed prior to a gateway to a network, but distributed network IDSs have been implemented for more complex networks. This also means that unlike host-based systems, the intrusion detection system executes in different hardware, as only network traffic is monitored. This keeps the monitoring hardware separate from what is being protected, thus most network intrusion detection systems would seem to fit best into the strict-hard security level. Despite this, however, it has been proven

that a number of network IDSs can be defeated by sending malformed packets and/or particular packet streams across the network which causes the detection mechanism to fail when such packets are analyzed [47]. As a result, many network IDSs only passively secure. Additionally, the separation of the hardware from what is being monitored also traditionally makes the implementation of network IDS easier and more scalable than host IDSs. This is because a network IDS can simply be placed on the network prior to the gateway to the network being protected, whereas a host IDS has to be integrated within a system and correctly interact with the entire system. It is this better scalability and ease of implementation that have made network-based intrusion detection popular in recent years.

Due to the ease of prototyping, using Field Programmable Gate Arrays (FPGA) to implement network-based intrusion detection systems makes up a large portion of the research in the network IDS field. As network-based IDSs only analyze network traffic, however, detection methods are limited primarily to signature-based detection. Not only does this limit the effectiveness of network-based ID, but this unfortunately means that much of the research into network-based intrusion detection systems, including those utilizing FPGAs, has mostly been limited to increasing the speed of pattern matching algorithms, which implement signature-based detection. This is due to having to keep up with ever increasing network transmission rates. Thus, there has not been much development into novel intrusion detection methods within this area of research. Additionally, the increasing use of encryption when transmitting data over

the internet is making network-based ID ever more difficult, causing a resurgence in host-based intrusion detection [11].

2.5 Virtual Machine Monitor-based Intrusion Detection

A virtual machine monitor (VMM) is an abstraction layer interposed between an operating system and the underlying hardware that supports it. The purpose of a VMM is to mimic the interface between the OS and hardware so that the VMM can monitor and control how an OS interacts with hardware. This allows the VMM to treat an entire OS as a separate thread of execution, thus transparently enabling the execution of multiple operating systems on the same hardware simultaneously and independently.

A few attempts have been made thus far to leverage the simultaneous and seemingly independent environment that VMMs can provide. Livewire, proposed by Garfinkel et al. in [17], and ISIS, proposed by Litty in [37], are two such efforts. Both systems treat the intrusion detection mechanism as a guest OS executing “simultaneously” with the host operating system on top of the VMM layer. The VMM layer serves as a common interface from which the guest OS can view state information of the host OS. According to Garfinkel, such an intrusion detection architecture combines the main advantages of both host-based and network-based intrusion detection systems - good visibility into the host’s state, while maintaining the security of the intrusion detection mechanism.

While VMM based systems can leverage these advantages to some extent, drawbacks still remain. First and foremost is the fact that the so called independence afforded by a VMM is almost entirely software-based, requiring the host and guest operating systems to have hooks into the VMM. Thus if the host operating system is compromised, the VMM as well as the intrusion detection system can potentially be compromised. Work done by King et al. has even resulted in a method known as SubVirt which implements a virtual machine-based rootkit (VMBR) [28]. Since a VMM already has more permissions than the OS it is protecting, a VMBR could be used to compromise not only the OS being monitored, but the VMM-based IDS as well. Another drawback to VMM-based IDSs is due to the VMM multiplexing the execution of multiple operating systems on the same hardware. As such, currently proposed VMM-based intrusion detection systems are not truly parallel in nature. Lastly, using a VMM partitions hardware utilization between any operating systems as well as the VMM itself. This can impose a large overhead when implementing a VMM-based intrusion detection system, and may make current proposals impractical due to an overall degradation in system performance.

2.6 Coprocessor-based Intrusion Detection

The goal of coprocessor-based intrusion detection, like VMM-based intrusion detection presented in Section 2.5, is to combine the visibility afforded by uniprocessor-based host intrusion detection while executing the intrusion detection system in a parallel and secure manner - the main difference being the use of a dedicated copro-

cessor to execute the IDS on hardware independent of the hardware/software being monitored. Coprocessor-based intrusion detection architectures can take many forms but are typically considered a form of host-based IDS as the coprocessor resides within the system that it is protecting (this is not a requirement). Work done on coprocessor-based intrusion detection is an active area of research, and represents the current state of the art in the intrusion detection field.

A handful of prototype coprocessor-based intrusion detection systems have been implemented to date. As certain coprocessor-based IDS implementations have some commonalities, we break up existing implementations into the following three groups described below.

- **Cryptographic Coprocessors:** These co-processors protect data by encrypting and decrypting information being transmitted between system devices, the CPU, and memory. Thus, if the data is intercepted somehow, the content cannot be compromised. These devices technically do not perform any intrusion detection tasks, but rather they ensure data integrity.
- **Add-in Coprocessors:** Add-in coprocessors monitor the state of the main CPU over a system bus such as the Peripheral Component Interconnect (PCI) bus. We term IDSs based on this implementation as “loosely coupled”. This is because, although such an implementation can monitor system state, the monitoring hardware does not reside at the same logical level as the main CPU, thus it has no way to exert control of the CPU in the event of an intrusion.

Furthermore, this also limits the amount of system state that can be gathered to what a system bus can access.

- **Integrated Coprocessors:** Integrated coprocessors reside at the same physical level as the processor it is monitoring. Thus we term them “tightly coupled”. IDSs based on such an implementation have the ability to exert control over the CPU being monitored. Additionally, these systems can access system state information at the CPU level, thus they are not only limited to state information that can be accessed via a system bus. Such systems have been enabled by the commercial availability of multi-processor systems in recent years, as well as multi-core processors even more recently. However, little work has been performed which explores how such architectures can be leveraged to aid security-related monitoring.

Using this categorization, we describe previously implemented systems that are relevant to our research below.

2.6.1 Cryptographic Coprocessors. While encryption is primarily involved in intrusion prevention and protection of sensitive data, we still believe that it is worth briefly mentioning. Implementing cryptographic coprocessors was the first foray into using a processor other than the host processor for security related tasks, thus paving the way for the development of co-processor based intrusion detection systems.

Cryptographic coprocessors can be used to encrypt and decrypt data sent within a system. This can ensure that if the data is intercepted somehow, be it by a mali-

cious process or through some other unintended means, the data cannot be accessed. Many cryptographic coprocessors are implemented as what has been termed “secure coprocessors”. These coprocessors have a dedicated CPU and access to dedicated non-volatile storage that can store vital information such as cryptographic keys, sensitive data, logs, etc. in a secure location. These coprocessors have been shown to be able to handle digital rights management, copy protection, and various e-commerce applications [39, 49, 74]. Research has also demonstrated that cryptographic coprocessors can even be used to make untrustable software, such as a standard standalone OS, trustable [23].

More recent uses of cryptographic processors have been to create a secure bus structure within the system. SECA, proposed by Coburn et al in [14], is an example of such a system that implements a secure bus structure. The cryptographic coprocessor is used to encrypt all data sent within the system and only components with the correct keys can decode that information. While this does not prevent an intrusion from occurring, it does ensure that data integrity is maintained in the event of an intrusion. Commercial availability of such a capability has recently been realized using the trusted platform module (TPM) [5]. Rather than be implemented as an add-in card in a system, the TPM is integrated into the system’s motherboard or Northbridge chip. Intel is using the TPM in its LaGrande security technology to provide cryptographic-based security for protecting sensitive data and peripheral communications that could lead to an intrusion [24].

2.6.2 *Loosely Coupled Monitoring Coprocessors.*

2.6.2.1 Independent Auditors. Molina and Arbaugh present a method of implementing independent auditors for file system integrity checking in [40]. As the paper's name implies, this system audits files to determine if an intrusion has taken place. The auditing work is performed by a coprocessor implemented on a PCI card in a standard personal computer architecture. The independent auditor (coprocessor) logs all changes to the filesystem and performs all auditing calculations to determine the integrity of the filesystem. Auditing is based on a policy file that defines what files are to be checked and what parameters are to be verified. The independent auditor periodically retrieves information pertaining to the files in question and checks them with the known good values stored in the independent auditor's local memory. The independent auditor can also keep secure logs of process activity, measurements, or other events. This can provide for a computer security forensics capability. The logs are stored in a trusted state which is ensured by the periodic file system integrity checks. If an integrity check results in an alarm, the data logged since the last known trusted state verification is considered to not be trustable.

As this system is implemented as a coprocessor that independently accesses the host system's filesystem, all auditing tasks are done in parallel as the host processor executes. This has little impact on host processor performance, however contention for the system bus is increased. Information audited is limited to only what can be gathered via the host processor's filesystem. Additionally, as the auditing of files is

periodic, the host is still potentially vulnerable to timing attacks. Despite this, by the nature of how the independent auditor is implemented and how it accesses state information, the IDS itself can be considered tight-hard secure.

2.6.2.2 CoPilot. The CoPilot system, developed by Petroni et al. and presented in [46], is a coprocessor-based IDS that monitors the integrity of a Linux-based kernel at runtime. This integrity monitoring is achieved by the coprocessor having visibility into the host processor’s physical memory space and looking for changes that are indicative of malicious activity. In the case of the CoPilot system, malicious activity is defined as the installation of known rootkits which can compromise the security of the host processor and the OS.

According to Petroni, there are six requirements that a coprocessor must meet in order to effectively monitor the integrity of a kernel at runtime:

1. Must have unrestricted memory access in order to view the host processor’s entire memory space.
2. The monitoring process must be transparent to what is being monitored.
3. The coprocessor must operate independently of the processor that it is monitoring.
4. The coprocessor must have sufficient power to process a large number of operations on memory.

5. Must contain enough memory resources to keep a consistent memory image of a non-compromised host
6. Must be able to securely report the state of the system via the use of a dedicated channel to an admin station.

In order to meet the above requirements, CoPilot uses a coprocessor that resides on a peripheral component interface (PCI) card. In so doing, the coprocessor can only receive data via the PCI bus. The PCI bus is afforded access to main memory through the system's memory controller which coordinates accesses made to main memory by the CPU and peripherals residing on the system buses. This allows CoPilot to monitor the production processor without there being any explicit communication between the processors themselves. As such, the CoPilot system falls within the tight-hard security category of our monitoring system security categorization.

Typically, for a device to access main memory, the device's address must be translated to a physical address in main memory that the device can then access. Interestingly though, due to the personal computer-based architecture of CoPilot, the PCI bus' address space has a one-to-one mapping to main memory. This allows the coprocessor to access main memory without the need to have the memory addresses translated, thus reducing the overall overhead associated with the coprocessor monitoring system memory. Once the coprocessor has access to main memory, it then monitors specific memory locations for changes to certain invariants. Memory locations of interest include locations containing kernel text or jump tables of kernel function pointers. As this does not look for specific symptoms of known rootkits,

but rather uses anomaly-based detection, this method may detect some previously unknown rootkits.

The CoPilot system does have some drawbacks, however. Most notably, CoPilot can only monitor memory locations that correspond to fixed pages, limiting monitoring to only those portions of the kernel hard-wired into physical memory. User processes cannot be monitored due to the dynamic (non-fixed) nature of the virtual memory subsystem employed in modern multi-programmed operating systems. Furthermore, CoPilot can be circumvented with sophisticated relocation attacks as well as timing attacks. This is due to the fact that CoPilot only monitors main memory and only does so every 30 seconds. Monitoring can not be performed faster than every 30 seconds, as bus contention becomes a limiting factor.

While 30 seconds may seem like a small window of time, it is large for a processor. For example, consider a superscalar host processor operating at a frequency of 1GHz with an average of 2.5 instructions per clock (IPC) - a very conservative configuration by today's standards. Within a 30 second time frame on such a system, 75 billion instructions on average will have executed! To put this in perspective, the SQLSlammer worm that was one of the most devastating Internet attacks of all time - it brought down 5 of the 13 Internet root nameservers - was only 376 bytes in size [15]. Assuming an average instruction length of 32 bits (the targeted x86 architecture actually uses variable length instructions), SQLSlammer contained roughly 94 instructions. Even with loops in the code and other processes executing for a portion of the CPU time, it can easily execute within the 75 billion instruction win-

dow. Although, it would have been detected eventually, it would still have caused the intended damage.

2.6.3 Tightly Coupled Monitoring Coprocessors.

2.6.3.1 CuPIDS. Williams' paper is one of the more recent implementations of a coprocessor-based IDS [59]. Rather than use a coprocessor located on a separate daughter card from the host CPU as with CoPilot and the independent auditors system, CuPIDS leverages the uniform memory access (UMA) multiprocessor model to perform intrusion detection and security policy compliance monitoring (SPCM). CuPIDS is implemented on a dual-processor system, although it can operate in any UMA-based multiprocessor/multicore system regardless of the number of processors. A single instance of FreeBSD executes in a symmetric multiple processing (SMP) fashion on the two cores, however the cores are leveraged by the OS in an asymmetric fashion - one core for production processes and the other for monitoring processes. As such, only one of the processors in the dual-processor system is available to the user for executing production code.

The tightly coupled nature of CuPIDS provides it with a very powerful capability - the monitoring CPU has access to virtual memory. As such CuPIDS can monitor code executing in both the kernel space as well as the user space, whereas CoPilot can only monitor code that resides in the kernel space (i.e., hard-wired pages). CuPIDS is afforded this ability by the coprocessor being at the same logical level as the processor executing the code being monitored. That is, the coprocessor in CuPIDS has all

permissions and capabilities of the production processor, whereas the coprocessor in CoPilot only has the permissions and capabilities of a peripheral within the system.

The CuPIDS architecture operates under the assumption that the operating system is not compromised. As only a single OS executes over multiple processors, this must be the case in order to ensure trustable operation of CuPIDS, as the OS itself houses the monitoring functionality. The backbone of the CuPIDS architecture are CuPIDS Production Process (CPP) and CuPIDS Shadow Process (CSP) pairs. A CPP is the process executing on the production processor core and its corresponding CSP is the process running on the shadow processor core that monitors that particular CPP. When a production process is to be monitored, a CPP and CSP are created and checked to ensure that they can be trusted. If both can be trusted, the CSP, followed by the CPP, are loaded into memory and “hooks” from the CSP into the production process’ virtual memory space are created. The CPP then executes on the production processor while being monitored by the CSP executing on the shadow processor.

In order to keep efficiency as high as possible, the CSP performs checks on its corresponding CPP only when certain events that can be used to detect an intrusion occur. Such anomaly-based events include variable use/creation/deletion events and checkpoint events - both of which are inserted into the CPP before execution. Events are communicated from the CPP to the CSP via streamlined system calls that are sent through the operating system’s kernel memory space.

In the instance where an event is triggered and an intrusion detected, the CSP can either simply notify that the CPP has been compromised, or it can notify as well as block further execution of the process - a function that the CoPilot System cannot perform [46]. Furthermore, events can be placed before/after key data structure modifications, in honeypot code (i.e., invariants), or randomly. Since these events can be placed anywhere and occur at any time, such an approach makes it very difficult, if not impossible, for a timing-based attack to compromise a CPP.

Furthermore, CuPIDS not only uses anomaly-based intrusion detection as mentioned above, but also specification-based intrusion detection as well. This is done through the use of white lists. When a CPP is created, a white list containing function, library, and system call source-destination pairs is created. Thus, when a jump or branch in the code is taken, the branch destination address is compared to the values in the white list to ensure that the branch is valid. This can also be used to perform stack monitoring in order to detect buffer overflow attacks.

Not only is the CuPIDS architecture robust in terms of capabilities, it can also detect intrusions in a matter of thousands of instructions rather than millions of instructions that uniprocessor host-based (i.e., software-based monitoring) methods typically take. Due to such a fast response time, the CuPIDS architecture affords the ability for self-healing. This is done by leveraging the knowledge that there are a number of known dangerous libraries that exist. When a function from such a library is called, the page that corresponds to the data to be operated on can be automatically copied. If the data is subsequently damaged by that function being

called maliciously, CuPIDS may be able to automatically repair the damage using the saved page, as if the malicious code had never executed. Furthermore, the fast response time of CuPIDS also allows the forensic logging of intrusion events for later analysis.

Despite CuPIDS' robust nature, it is not without its drawbacks. The most notable of these drawbacks is that despite efforts to minimize overhead in CuPIDS, there is still roughly a 15% performance decrease compared to non-monitored execution. Also, since the OS is executing in an SMP fashion on both the production CPU and the shadow CPU, the shadow CPU is not invisible to the production CPU. Thus, if the OS is ever compromised, the entire intrusion detection system is susceptible to being compromised. As a result, CuPIDS falls within the self-secure monitoring system security category. Moreover, it may not even be verifiable whether or not the intrusion detection system has been corrupted if the OS is ever compromised.

2.6.3.2 A Security Enhanced Chip Multiprocessor. Shi et al. propose an IDS that is very similar to the CuPIDS architecture in [50]. Like the CuPIDS architecture, the Security Enhanced Chip Multiprocessor (SECM) uses multiple processing cores in an asymmetric manner where one core is used to monitor the host core in parallel. Unlike CuPIDS, however, the SECM uses two distinct operating systems - one executing on each processing core. Furthermore, the SECM uses a shared Level 2 (L2) cache, with each processor core containing its own Level 1 (L1) data and instruction caches. State information is gathered by tapping directly into the memory

bus between the caches and the processor core itself as well as instruction trace data. Thus, each time a monitoring processor requests data, the request is checked by the SECM thus enabling the monitoring of memory via cache, not main memory itself.

The operating systems of the SECM are implemented in a unique way for a multi-core processor. The OS on the host processor is just a normal Linux-based kernel. The OS on the monitoring processor is a stripped down Linux-based kernel. This is done to give the monitoring OS a smaller “footprint”, thus making it more secure. Furthermore, Shi et al. have implemented a scheme that partitions the privileges of the two operating systems. The host OS is given privileges to only a certain amount of non-threatening high level functions, while the secure OS is given full privileges over the system. This ensures that the monitored processor core cannot circumvent the secure processor core in the event that the monitored core is compromised. As a result, the SECM falls within the tight-hard monitoring system security level.

As with CuPIDS, the SECM architecture uses many forms of detection to determine if an intrusion has occurred. It implements both anomaly-based and specification-based intrusion detection via checkpoints and call/return address checking. State information is gathered via the memory bus between the L2/L1 caches and the processor core itself. The execution trace is also used. Monitoring is performed by the privileged processor checking each request made to memory by a lower privileged processor. Such state information is gathered from the local memory bus connecting the caches to the processor core rather than the actual memory bus external to the processor, enabling the detection of relocation attacks that occur in cache. Further-

more, kernel-level rootkits can be detected via openly available tools. In the event that a rootkit is successfully installed, the SECM can recover, as a clean copy of the monitored OS kernel space is stored by the privileged processor.

Performance of the SECM is somewhat unclear, however. Unlike the CuPIDS architecture, the SECM is not actually implemented in hardware to date. Shi et al. rely on performance emulation by a simulator to determine a general estimate of performance. Furthermore, there is no comparison to the performance of a similarly configured, non-monitored system. Thus, the benefits afforded by monitoring using the SECM are potentially marred by performance degradation.

2.7 Hardware-assisted Security Mechanisms

While the research we have described up to this point has focused on actual intrusion detection systems, many hardware-based mechanisms have been proposed the focus specifically on a particular security threat or a small subset of threats. Hardware-assisted security mechanisms are usually intended for specific applications. As such they have thus far been targeted more at embedded and application specific markets that tend to have tighter design constraints and more static software environments. Implementation of such hardware-based mechanisms also tend to apply to computer architecture in general, thus the concepts are not limited to a particular system structure. This section describes such security mechanisms that are either related to our work or have served as a foundation for the direction that we have taken.

2.7.1 Hardware-based Stack Protection. One of the most common attacks used to compromise a system is known as stack smashing [45]. This attack uses some weakness, usually a buffer overflow exploit, present within the code to rewrite information residing on the stack. Information is rewritten in such a way as to rewrite the return address of a function that has yet to complete. When the function completes and attempts to return to the location from which it was originally called, it uses the address that was rewritten by the buffer overflow, causing the control flow of the executing process to be redirected from its own code to malicious code injected onto the stack or residing somewhere else in memory.

In order to defend against the type of attack described above, a secure return address stack (SRAS) has been proposed and simulated by Lee et al. in [34]. The SRAS is a hardware-based last-in, first-out (LIFO) buffer similar to a stack, however it only stores return addresses of functions whose blocks have been pushed onto the stack rather than an entire function block. When a function returns, the address stored at the top of the SRAS is then compared to the return address stored in the main stack. If the two addresses differ, then the processor is notified so that it can take appropriate action. This is realized via adding special instructions that control the operation of the SRAS. As such, monitored code must explicitly communicate with the monitoring mechanism to function. This, combined with its hardware-based implementation, places this security mechanism in the loose-hard monitor security level.

Although the hardware-based nature of the SRAS makes it more secure, it is not without its drawbacks. One such drawback is that it can only store a finite number of return addresses. Thus, if the stack becomes completely filled, then the contents must be moved to main memory to make room for new return addresses to be placed on the SRAS. Memory locations containing overflow return addresses are protected by only allowing the kernel to access them (assuming the kernel has not been compromised through some other means). Furthermore, it should also be mentioned that although the SRAS is dependent on the LIFO nature of the stack, it can handle non-LIFO control flow in a number of ways. These include: not-allowing non-LIFO code to be executed, creating new commands within the compiler to push return addresses onto the SRAS at times other than function calls, and deactivating the SRAS completely.

2.7.2 Microinstruction-based Monitoring. Microprocessors each have an instruction set architecture (ISA) that is dictated by the architecture of the processor. The ISA defines the machine-level instructions that allow the user/programmer to control the hardware. Most microprocessors today also implement microinstructions that coordinate data and control flow within the processor. Thus, a single machine instruction can be composed of many microinstructions. Such microinstructions are not accessible by the programmer and enable the modification of a processor's architecture while keeping the same ISA for compatibility reasons.

Rather than add external hardware to monitor the execution of a processor, Ragel et al. have proposed a method of creating self-checking instructions by modi-

ifying the microinstructions that implement potentially “dangerous” instructions [48]. Ragel’s system is intended to be applied to embedded applications only, as their proposed methods also require a modified compiling chain in order to determine which instructions in code are deemed as “critical”. Buffer overflow attacks, fault injection attacks, and out of bounds memory address accesses are checked for. Their proposed microinstruction changes that implement these checking schemes are just used as examples of what can be done with their system and do not represent the full capabilities of such a system.

Their system prevents certain buffer overflow attacks by using a hardware-based return address stack like that described in 2.7.1. Faults injected into the instruction path are checked by reading the instruction memory before an instruction is fetched and comparing that instruction with the one that is fetched by the instruction fetch unit. Faults injected into the data path (i.e., the execution pipeline) are checked by storing the write-back address (determined in the instruction decode stage) to a FIFO buffer and comparing that to the actual location where the data is written back to during the write-back stage of the pipeline. Memory boundary checks are also performed by making sure that instructions do not access areas of memory outside of a particular range. This method is rather coarse-grained, however, and finer-grained methods are described in 2.7.3.

Clock speed reduction as a result of implementing micro-embedded monitoring is reported to be less than 7% for all applications tested. Area overhead associated with the added microcode is also relatively minimal at no more than 15% added area.

It should be noted, however, that no mention is made of how effective these techniques are at actually detecting the attacks that they are designed to defend against.

2.7.3 Control Flow Monitoring. While tracking the execution flow of code is not a new concept in the ID field, doing so at the hardware level is a relatively recent development. As code is ultimately executed by the processor as machine level instructions, it is logical to assume that we can gain insight into the execution of the code if we could directly monitor the pipeline of a processor. Some research has been conducted to view this state information and utilize it for intrusion detection tasks.

In [72], Zhang et al. propose modifying the XOM secure processor model to be able to check the control flow of a program for anomalous events using hardware-based methods. Whereas software-based control flow monitoring techniques typically can only track control flow at a function/syscall granularity, Zhang’s method can track a program at the instruction level. For the detection system to know what what is considered “normal”, two methods are used. The first consists of parsing the text segment of the process to be monitored. This determines where all branching instructions reside in the process’ virtual memory space as well as the address that each branch can branch to. The second method involves executing the process in a known trusted state in order to train the detection system to be able to recognize what branching behavior is considered “normal”. With this information, the detection system monitors the current program counter, the next program counter, and the type of instruction as the processor executes. The detection hardware is implemented as

a staged checking path where the more common branches are checked earlier in a previous stage in order to keep pipeline stalls to a minimum. If a branch jumps to an address not deemed as legitimate or a branch instruction occurs where there should not be one, then an exception is raised and handled securely by the monitoring hardware. A similar action is taken if a degree of abnormal branching behavior occurs. A hardware based return address stack like that described in 2.7.1 is also implemented to ensure that function and system calls return to the appropriate address when complete.

Further work done by Zhang et al. and presented in [73] has improved on the anomaly detection capabilities of their previous work. Their previous work could only look at a single branch when checking for anomalous behavior. However, their updated technique can now correlate multiple branch instructions when checking for anomalous control flow behavior. This is done by recording all “normal” execution paths during training. Such paths are stored in a table that can be accessed by the control flow checking hardware during runtime. This anomalous path detection is not limited to a particular number of branches (control flow changes). Furthermore, this more recent work has also improved direct jump checking by parsing dynamic libraries linked at runtime in addition to the already parsed process binary (text segment). Results of this work show that anomaly detection can be done with very little overhead due to the control flow checking being done in hardware. Additionally, anomaly detection efficiency was found to be high and occurred within a few cycles of entering the production processor’s pipeline due to the staged design of the control

flow checking hardware. Drawbacks still exist, however. These include the detection efficiency's dependence on how well trained the system is and the fact that system must be trained for each process that may be monitored - a possibly arduous process for a dynamic, multiprogrammed environment. As such, these methods are probably better suited to embedded applications where the software environment is more tightly controlled.

Arora et al. present even more recent research efforts in [3] whereby they introduce a mechanism that provides for multi-grained, real-time monitoring at the instruction execution level. To enable this, the program counter and instruction register are used to expose the current executing instruction and its corresponding address in memory to the monitoring hardware. Detection is accomplished by utilizing specification-based ID techniques. However, it should be noted that no security policy is explicitly defined by the developer/user. The specification is created by utilizing static program analysis techniques to define permissible behavior which is then checked against during program execution. Additional static analysis is also performed at program loading to gather address information for dynamically loaded libraries which is unknown at compile time. The program attributes that are extracted to define the specification are described below:

- **Inter-procedural Control Flow:** This attribute is concerned with proper control flow between different functions within the code being monitored. This information is extracted from the code by creating a function call graph that maps

all function calls and their return addresses within a program. The function call graph is then converted into a finite state machine (FSM) that is implemented in hardware. Checking done using this information is the most coarse-grained of all of the invariant checks.

- Intra-procedural Control Flow: This attribute is concerned with proper control flow within a function residing in the code being monitored. This information is gathered by determining all possible branch source-destination pairs within all functions within the code to be monitored.
- Instruction Stream Integrity: As not all attacks change the control flow of the targeted program, this attribute is used to ensure that the code within a basic block has not been modified. These invariants are determined by creating a hash value for every basic block within the code. Checking done using this information is the most fine-grained of all of the invariant checks.

The mechanism is implemented in three main blocks - one corresponding to each of the three types of invariant checking performed. Intra-procedural control flow checking is performed similarly to hardware-based return address stacks described in 2.7.1. The FSM mentioned earlier compares the state index (generated at compile time) of the start and return addresses. If the start and return state indexes correlate to an allowable control flow change represented in the FSM, the control flow change is allowed to continue. Inter-procedural control flow checks are performed by using the starting address of the currently executing function to calculate offsets of the branches within the function as they execute. These are then compared to the stored branch

source-destination pairs (generated at compile time). Instruction stream integrity is accomplished by buffering the instructions of a basic block as the processor executes them. When a branch instruction is reached, a hash of the instruction buffer is computed and compared to the corresponding hash that was statically generated at compile time.

Each invariant check includes a mechanism for stalling the processor should any part of the detection process occur too slowly. As such, this detection runs in lock step with the executing code being monitored. The detector's state is managed by control logic in each of the three main blocks. As such, this detection mechanism does not execute any software of its own. Due to the parallel, hardware-based nature of this mechanism, little overhead is introduced. The only degradation in performance is caused by the hashing of basic blocks for instruction stream integrity checking. In this case 50%-60% of basic blocks are reported to be able to be hashed without having any noticeable performance penalty. It should also be noted that since this mechanism is synthesized based on static analysis techniques, this mechanism is application/process specific.

2.7.4 Non-executable Memory. Certain forms of malicious attacks execute code from a process' data memory space - a space typically used for storing only data. Such attacks are enabled by the fact that processors based on a von Nuemann architecture have a shared data and instruction memory space. As a result, the processor can not distinguish a data access from an instruction access. Recently, however, mod-

ern von Nuemann-based processors have been updated to be able to distinguish the difference, disallowing instructions from executing if they resided within a portion of memory deemed as non-executable. AMD first commercially introduced this technology, known as No-Execute (NX) bit, and later Intel with the eXecute Disable (XD) bit [56].

The NX/XD bit works by adding an extra bit to all addresses within the page table entry (PTE) [19]. If the program counter is set to (i.e., branches to) an entry in the page table that has the NX/XD bit enabled, the instruction is not allowed to execute and an exception occurs. Thus certain attacks, like buffer overflows, can be prevented quickly and efficiently in hardware.

2.8 Trusted Operating Systems

While our work does not focus on implementing a trusted operating system, it does require that the software or operating system to be protected can execute in a trusted state for certain tasks. Most operating systems start up via an unrestricted process. That is, that there are no checks to ensure that what is being booted has not been compromised in some way. Lipton et al. propose a method call Spy that can create a trusted environment from untrustable machines [36]. Through their research, they formally define the problem of trusted software and prove that in order to be able to trust software, some form of hardware - the spy - must be present to enforce certain key policies. Similar work is presented in [23]. This research specifically explains how an actual coprocessor can be used to perform invariant checking of key kernel data

structures, ensure filesystem integrity, as well as detect viruses - all of which work together to create a secure operating environment. Earlier work by Arbaugh et al. resulted in the Aegis system which implements system wide modifications to ensure that an OS can be booted into a known trusted state [2]. Modifications to the system include creating BIOS enhancements that allow for a multi-level booting approach where each level in the boot process provides for more privileges once the previous level has been successfully completed.

Perhaps most relevant to our work, however, is a method used by CuPIDS by which the OS operates in an untrusted state, but certain key operations require that a temporary trusted state be created in order to complete. A key example of this is the creation a shadow process to hook into and begin monitoring a production process. Once the shadow process is created and is executing on the shadow CPU, the OS is no longer in a trusted state. Further information on how the secure startup of a trusted process completes can be found in [59].

III. Research Concepts and Methodology

In this chapter we discuss the concepts that serve as a basis for our research. We begin by presenting our research hypothesis as well as discuss the security exploits that our work is intended to target. This is followed by the general architecture - hardware and software - that enables robust parallel security monitoring at the hardware level. We then go on to present the general functional primitive concepts that leverages our architecture to gather and process state information.

3.1 Research Hypothesis

If there is one main drawback (with respect to security) in the development of modern computing architectures, it is that they have primarily been designed with performance, rather than security, in mind. Our research explores ways in which we can break through the limitations that current architectures impose. We intend to define new means by which system state can be revealed and processed to allow for more robust and flexible security policy compliance monitoring mechanisms. To accomplish this, we believe a contemporary multiprocessor computer architecture can be modified in such a way as to allow the creation of functional primitives that can expose and process state information in ways previously unavailable at the hardware level. This will not only allow for more secure, better performing, and more capable security policy compliance monitoring, but also provide a flexible architecture by which security functions can be tailored to particular applications on the same platform.

3.2 *Targeted Exploits*

Our theories apply to a very broad range of events. As we are viewing events at the hardware level, both malicious activity as and erroneous program behavior can be detected. As such, our concepts can be viewed as applying to SPCM. SPCM includes detecting malicious activity as well as detecting unexpected activity, such as bugs or errors that software developers and quality assurance (QA) testing did not catch. Therefore, our research is not trying to defend against any one specific subset of events. Additionally, it should be noted that the memory introspection techniques we propose focus on main memory; however, it is safe to assume that the monitoring concepts we propose can also provide us the ability to monitor a processor's cache. Thus, the monitoring concepts resulting from this research could be applied in such a way as to protect against attacks that leverage the cache (i.e., relocation attacks).

3.3 *Architectural Overview*

This section provides an overview of the hardware and software architecture of the platform for which the functional primitives are being developed for. It also describes the abilities and features that this architecture architecture enables.

3.3.1 Hardware Architecture. The general architecture of our parallel monitoring concept is shown in Figure 3.1. As can be seen, the architecture contains multiple processing elements - the production processor unit (PPU) and the shadow monitoring unit (SMU). The PPU is responsible for executing all user-related code

such as the system's main OS (if one is present) and any applications. The SMU is responsible for performing all security monitoring tasks. The number of processors the architecture supports is unlimited, however, each PPU in the system should have a dedicated SMU that corresponds to it.

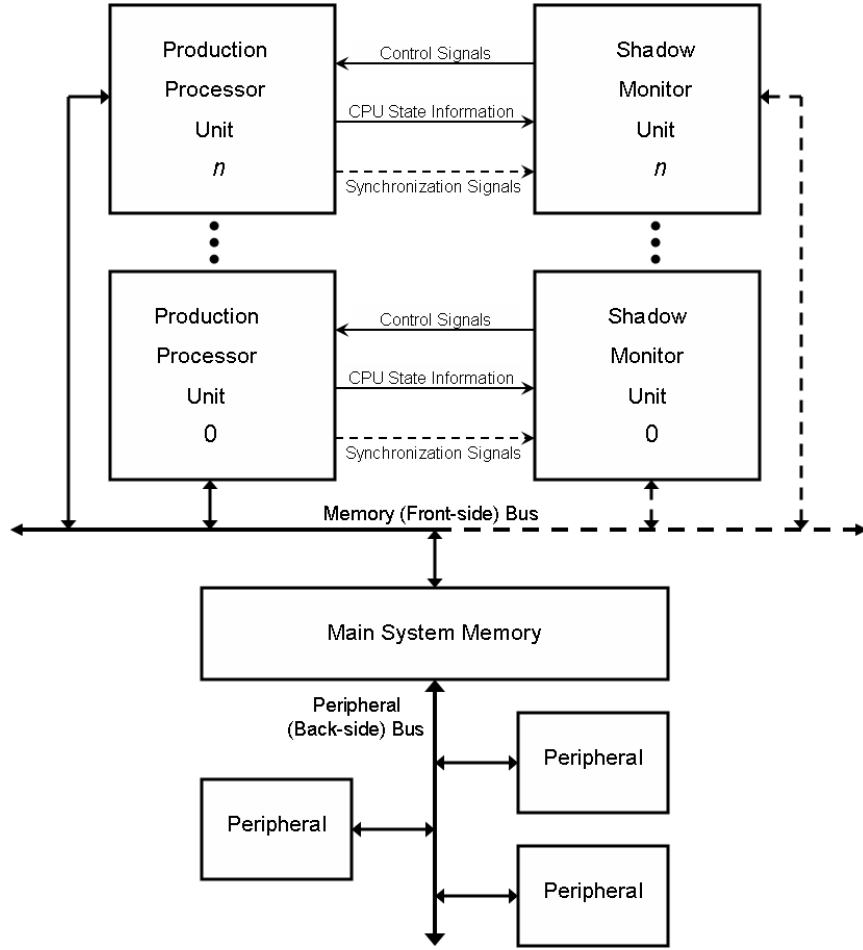


Figure 3.1: General Overview of the Hardware Architecture

A shared memory architecture in a UMA configuration is used as the foundation for developing our parallel security monitoring techniques. This allows both the PPU and the SMU to have the potential to access all memory and peripherals within the system. Additionally, the system is configured in such a way as to restrict access

to certain peripherals for each of the processing elements. An example would be to assign the serial port of the system to only be accessed by the SMU for administrator control purposes. This peripheral assignment is dependent on the application and controlled by the system designer.

Although shared memory multiprocessing platforms have been available for years, shared memory multicore processors (multiple processing cores residing within the same physical package) have become commercially available only recently. Multicore chips have the potential for their cores to be more tightly coupled than the cores of their multi-processor counterparts. This is because the cores of a multicore chip can be designed to communicate with each other via on chip facilities such as on-die interconnects or a shared cache, rather than having to rely on an external (to the entire processor) system bus. Our research leverages such a multicore architecture to enable new forms of parallel security monitoring. As a result, we can tap into certain signals within the PPU, enabling the PPU to transmit state information to the SMU purely in hardware as depicted in Figure 3.1. This eliminates the need to use the system bus for communicating the state of the PPU to the SMU, allowing our monitoring techniques to not be limited only to state information accessible from the memory bus and/or the core's debug logic. Similarly, this tighter coupling of the cores allows control signals from the SMU to the PPU to be implemented without the need of an external system bus as well.

3.3.2 Shadow Monitoring Unit Configuration. The SMU in the architecture can be implemented in a number of ways providing a level of flexibility for efficient and capable hardware-based, parallel security monitoring. Different types of monitoring requires different types of hardware. For example, some mechanisms may need to detect illegitimate events immediately, thus requiring the monitoring mechanisms to be implemented using real-time-logic (RTL). Additionally, a mechanism may need to perform a complex algorithm on state information over a period of time, in which case a coprocessor would be more appropriate for implementing the monitoring mechanism. As such, the SMU can be implemented as a coprocessor, RTL, or a combination of the two. Furthermore, if implementing the SMU using a coprocessor, it can either be identical to the PPU or a totally different architecture altogether. As such, the SMU can be seen as a black box in an overall architecture that we propose to be well suited for creating and implementing real-time, parallel monitoring mechanisms. The configuration of the SMU is left to the system designer and is dependent on the needs of the particular application(s).

How the SMU is implemented affects the capabilities of the monitoring hardware, however. If a coprocessor is used, the SMU can have native access to main memory and can execute actual code. If RTL is used to implement the security mechanism(s), however, the SMU will lose the ability to execute code. Additionally, using RTL to implement the SMU will make accessing main memory less trivial (but not impossible) than if implementing the SMU as a coprocessor. However, it is not required that the SMU have access to main memory depending on the application of

the SMU. As such, the SMU may not always take advantage of the shared memory nature of our proposed platform. To denote the optional nature of the SMU's connection to main memory, the connections from main memory to the SMU are denoted by dashed lines in Figure 3.1. It should also be noted however, that no matter the implementation of the SMU, the SMU will always have direct connections to the PPU for gathering state information and sending control signals.

3.3.3 Software Architecture. Shared memory architectures are typically implemented to facilitate an SMP environment. In an SMP system, every processor is exactly the same and executes similar types of tasks. For this to occur, an OS must be able to support SMP so that the OS can assign tasks that can take advantage of multiple processors. SMP systems also rely heavily on dependent tasks using the shared memory nature inherent in an SMP system to share data among the multiple processes on different processors.

Although the hardware architecture is built upon a shared memory architecture, the processors are used in an asymmetric manner; that is, one processor is responsible for performing entirely different tasks than the other processor in the system. Rather than use a single OS spanning both processing elements, two entirely separate OSs are used - one executing on each processor. Whereas the CuPIDS architecture relied on the OS for gathering state information and communicating between the processing elements, we are able to use two separate OSs since we gather state information and perform inter-processor communication via hardware. This also enables our system

to be implemented on computer architectures that do not support cache coherency, as cache coherency is a requirement for SMP systems. The software architecture is shown in Figure 3.2 below.

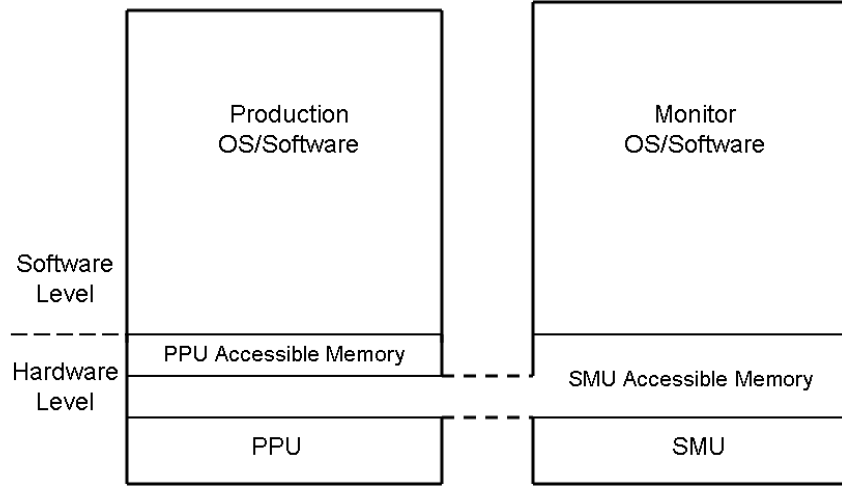


Figure 3.2: General Software Architecture

Despite both OSs being standalone and using hardware as a communication medium, the PPU OS may still need to be modified in order to explicitly send certain synchronization signals to the SMU. This explicit communication is depicted by the dashed arrow in Figure 3.1. This is dependent on the the type of monitoring being performed. However, as we desire the operation of the SMU to be as transparent to the SMU as possible for security reasons, explicit communication from the PPU to the SMU is kept to a minimum. An example of a modification required to implement such a communication mechanism is to modify the scheduler to trigger an interrupt signal on the SMU to notify the SMU that a context switch has occurred. Thus, there may be minimal coupling between the PPU code and the SMU code.

It should be noted that an OS need not be implemented on either the PPU or SMU. The implementation of software is completely up to the system designer and is dependent on the type of security monitoring to be performed as well as the target computing environment (i.e., an embedded system may not use an OS, but a general purpose system will always use an OS). This creates a large degree of flexibility in how the security-related monitoring is implemented and to what systems our parallel monitoring techniques can be applied.

3.3.4 Monitoring System Security. One of the ultimate goals of our research is to create a real-time detection system with access to state information, while exposing as little of the monitoring mechanism to the PPU as possible. We assume the PPU is vulnerable to attack, so the less visible the SMU is to the PPU, the more secure the SMU will be from attack. The software executing on the PPU is the primary medium of attack we are protecting the monitoring system (i.e., the SMU) against. Therefore, the software coupling between the PPU and the SMU is minimized as much as possible. As a result, hardware must be used to tightly couple the PPU to the SMU. This improves security over more software dependent coprocessor intrusion detection systems such as CuPIDS [59]. Additionally, the user only has visibility of the PPU in the system and as such has no way of explicitly communicating with the SMU. Thus, only a system administrator can directly communicate to the SMU and such communication does not go through the PPU in any way.

As the software executing on the PPU may have to be modified in order to explicitly coordinate with the SMU depending on the monitoring mechanism being implemented, coupling between the production code and the monitoring code is inevitable. Although the modifications would be minimal (synchronization signals at most), it still creates an avenue for an attacker to alter the operation of the SMU. However, since modifications should be minimal, the attack surface is reduced, and hence the portion of the PPU code that must be protected to ensure that the operation of the SMU cannot be illegitimately altered is decreased. Considering the amount of useful state information we are gathering, this is an improvement in the security of the monitoring mechanism itself compared to other host-based and coprocessor-based intrusion detection systems. It should also be mentioned that the need for the PPU to explicitly communicate to the SMU mostly affects a multiprogrammed environment where multiple different processes may need to be monitored.

In certain cases the PPU may need to send certain data regarding a specific monitored process to the SMU when such a process is created. An example of this is the page directory address of a process to be monitored. When this communication between the processing elements occurs, the PPU should be in a known safe state and only required portions of the kernel should have access to the communication mechanisms. This is similar to how a shadow process was created in CuPIDS [59]. Further information on creating a trusted OS state in an insecure system is described in [23]. As our research focuses on the security monitoring mechanisms themselves,

not on creating a single contiguous security monitoring system, this is outside the scope of our research.

3.4 Target Environment

Although coprocessor-based intrusion detection systems are more flexible than purely hardware-based security mechanisms and can be implemented in multiprogrammed (e.g., desktop and server) environments, they have their limits. In the case of a system like CuPIDS, the reliance on software makes the IDS itself vulnerable to attack and introduces communication overheads. In the case of a system like CoPilot which relies more on hardware to perform security monitoring, visibility into the system’s memory space is limited, and as a result, so is its flexibility. Our research focuses on further bridging the gap between software and hardware mechanisms in order to make security mechanisms that are high performing, yet flexible and secure. As such, we are not targeting one specific computing environment, but rather the entire spectrum of computing environments. The techniques we propose are just a sample of what can be done with the novel multicore shared-memory architecture we propose.

3.5 Functional Primitives

In this section we present models by state information is gathered from the PPU and leverage that information for security-related monitoring. A number of methods are presented here. Most methods are mutually exclusive of the others presented,

thus a single concept or a combination of all the concepts can be implemented into a design depending on a user's specific security needs and the target environment.

3.5.1 Multi-context Hardware Monitors. A number of hardware-based security mechanisms discussed in Section 2.7 are limited by being application specific. As such, only one instruction stream context can be monitored by these security mechanisms. This limits the effectiveness of these monitors in multi-programmed environments. While such hardware-assisted security mechanisms may be feasible for application specific and focused embedded applications, embedded applications are becoming more robust and complex. Additionally, these security mechanisms would also benefit common multi-programmed (multi-context) environments. It is important to mention that not every context needs to be monitored, but currently there is no way for the security mechanisms to even discern between separate processes. Thus, without running behavioral analysis on the system to create an idea of what behavior is acceptable, these mechanisms will not work correctly when trying to operate in a multi-programmed environment. Even with having done behavioral analysis, such monitors are only feasible for less complex or tightly controlled systems and/or are prone to producing false positives. Additionally, such mechanisms will have difficulty even if only trying to monitor a single process as the monitor cannot precisely and efficiently discern between different processes.

As we desire the SMU to be able to discern between different processes executing on the PPU in real-time, exposing the process identifier (PID) is the most logical piece

of state information that can enable this. In an OS, the system scheduler determines when each process executes. It keeps track of all processes and the state of the process using a table stored in the kernel space of the OS itself. This portion of the kernel is paged, but hardwired into main memory, thus the location where the currently executing PID is stored remains constant. Monitoring this location in memory can allow us to keep track of what process is currently executing.

It should be noted that we are not concerned with how the monitoring hardware knows which process or processes it needs to monitor, however, we are concerned about whether the capability exists to use the PID to allow the SMU to be able to discern between various executing processes. Once the PID has been made visible outside of the PPU, the SMU can take the appropriate action to enable multi-context hardware monitoring. There are three general ways this can be done and each is described ahead.

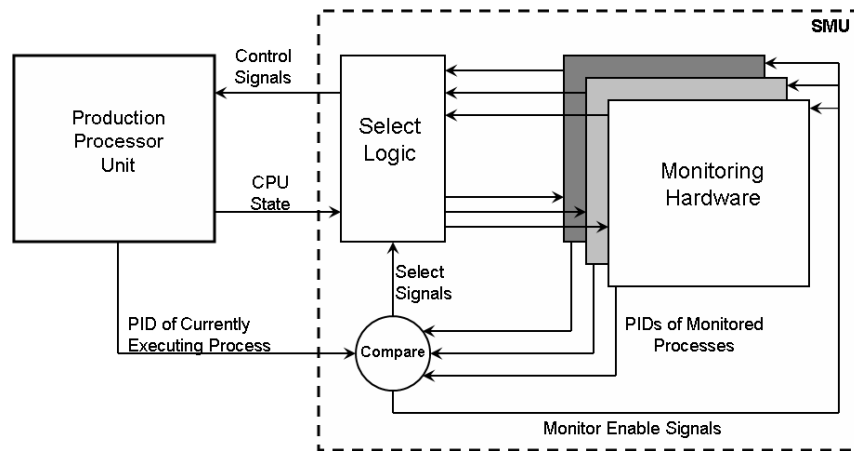


Figure 3.3: Multicontext Monitoring With Multiple Monitors

1. Monitoring a single process out of many currently executing processes: This is the simplest case. It requires the monitoring hardware to be enabled only when the PPU outputs the PID corresponding to the process to be monitored.
2. Monitoring multiple processes with the same hardware: This is similar to the first method, but the SMU has a list of PIDs that correspond to monitored processes. This case also requires some method of storing state when switching between processes being monitored. As such, it is practical to implement the SMU in the form of code executing on a processor core for the ease of writing to memory. Doing so also provides for flexibility by using the same monitoring hardware, but with different monitoring algorithms for different monitored processes.
3. Monitoring multiple processes with multiple types of monitoring: This is the most complex case. This method must monitor multiple processes, but uses multiple independent hardware monitors. Some form of selection logic is needed to generate the select signal that enables a particular monitoring mechanism, as well as complete the connections between the PPU and the SMU so the active monitor can retrieve state information from the PPU. This method is depicted in Figure 3.3.

3.5.2 Program Counter and Instruction Trace Exposure. The program counter (PC) keeps track of the memory location containing the currently executing instruction. We intend to monitor the PC as the PPU executes and use it to aid in

SPCM tasks. This information can be leveraged to provide the two main capabilities described below.

3.5.2.1 Execution Policy Enforcement. By keeping track of the PC, we can keep track of exactly from where in memory an instruction is being executed. Thus, when having defined where in memory code is allowed to execute and where it is disallowed, the PC can be used to check if such a policy is being adhered to or not. This knowledge of what code is allowable can have multiple granularities ranging from the global level to the basic block level.

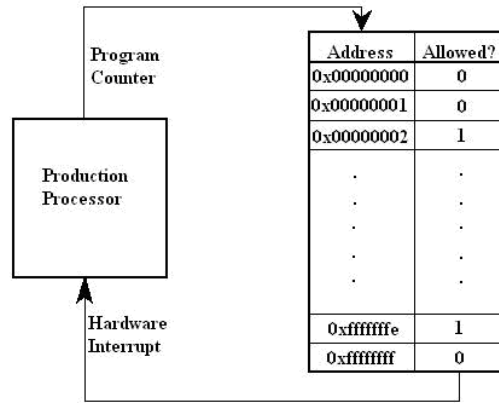


Figure 3.4: Program Counter Monitor High-level Architecture

Knowing the PC at any given time enables an ability similar to that of XD bit and NX bit technology from Intel and AMD, respectively. This technology prevents the execution of instructions residing in memory locations deemed as non-executable by adding an extra bit in the page table entries stored in memory [56]. Gaining access to the PC can provide a similar protection for processors that do not natively support it. This concept is depicted in Figure 3.4. For every instruction, its PC is

automatically checked by the SMU. Theoretically, we have the ability to keep track of every address in the memory space and then compare the PC in order to check if the executing instructions are allowed. Additionally, it should be mentioned that such a mechanism can also be used to aid in the enforcement of certain permissions in an asymmetrically shared memory space like the one proposed in Section 3.5.4.1.

3.5.2.2 Branch Source-Destination Address Checking. A number of researchers have proposed methods that monitor control flow changes as code executes for security purposes [16, 29, 34]. Furthermore, a number of hardware-based mechanisms have already been implemented to leverage such information for branch source-destination address checking [3, 73]. While not proposing any new methods to perform branch source-destination checking, its important to mention that the current platform can be used to implement such a system, as these techniques are an application of exposing the program counter and instruction state information. As such, our architecture can facilitate similar monitors.

3.5.3 Peripheral Access Control. Just as processes should only execute instructions from legitimate memory locations, processes should also only access the peripherals within the system that they were originally intended. As a result, a hardware mechanism that could enforce such policies would be beneficial. Moreover, implementing such a mechanism in hardware would make circumventing such a policies more difficult than if protections were implemented in software.

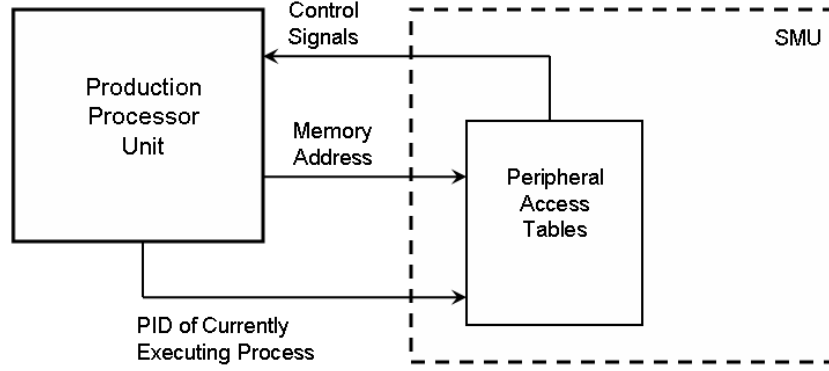


Figure 3.5: Peripheral Access Control Monitor General Architecture

The general architecture of the peripheral access control monitor is shown in Figure 3.5. As this primitive specifically monitors peripheral access on a per process basis, this requires techniques proposed in Section 3.5.1 to reveal the PID of the currently executing process to the SMU. Additionally, to determine what peripheral, if any, is being accessed by a process at any one time, the monitor also requires visibility into the addresses that the process is accessing. This assumes that the PPU uses memory mapped I/O, as communicating with peripherals occurs as read and write operations to specific address ranges via the main memory bus being monitored.

3.5.4 Hardware-based Memory Introspection. Various memory introspection techniques have been used for hardware-based security monitoring in systems such as CoPilot and CuPIDS [46, 59]. We present a number of techniques related to the hardware-based memory introspection below which are complementary to the primitives we have proposed thus far.

3.5.4.1 *Asymmetrically Shared Main Memory.* As our platform is con-

structed on a shared-memory architecture, both the PPU and the SMU can access the same main memory. This is ideal from a monitoring standpoint as both the PPU and the SMU can have visibility into the same physical memory space. Traditionally, however, all processors in a shared-memory architecture have access to the entire memory space and can read and overwrite the data corresponding to a process executing on another processor. Although safeguards are usually put in place (within the OS) to ensure that a process cannot alter another process, this cannot be assumed as true if the system is ever compromised. To limit this vulnerability, we wish to minimize the amount of knowledge the PPU has of the SMU, making the SMU's operation as transparent to the PPU as possible. Thus, the traditional shared-memory model must be altered to facilitate such a capability. This change is further reinforced by the architecture executing independent and different software on the SMU than the PPU, whereas a traditional shared-memory model is implemented with a single OS spread across multiple processors.

Figure 3.6 depicts a high-level view of how the shared memory is organized (not drawn to scale). In order to make the SMU's memory space invisible to the PPU (so the PPU can not be used to compromise the SMU), the PPU's software/OS must be instructed to view only a portion of the total available physical memory space. This region of memory should be contiguous, otherwise the PPU would have to be aware of the SMU's memory space - something we are trying to avoid. The SMU attains visibility into the PPU's memory space by being configured to have access to

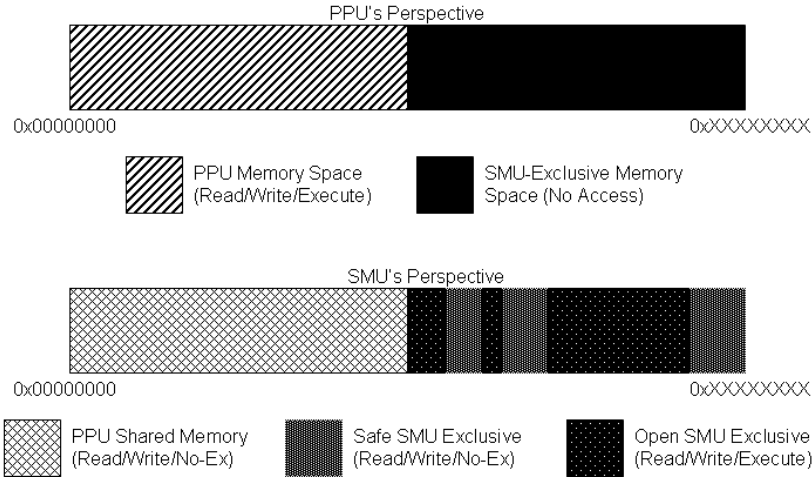


Figure 3.6: Memory Map and Permissions as Viewed by the PPU and the SMU

the entire physical memory space. As such, the SMU has access to its own memory space while still being able to access the PPU's memory space for security related monitoring. Essentially, a quasi non-uniform memory access (NUMA) architecture is created from a UMA architecture. This architecture differs from a traditional NUMA, however, in that the memory space is asymmetrically distributed, as the SMU has access to both the PPU's memory space as well as its own, while the PPU only has access to its own memory space. Additionally, a processor must request access to another processor's memory space in a traditional NUMA architecture, while our quasi NUMA architecture specifically avoids this requirement in order for the SMU to be as invisible to the PPU as possible.

While this architecture can provide the SMU visibility into the PPU's physical memory space, this is not enough. Further modifications to the traditional shared-memory model must be made in order to protect the integrity of the software executing from the PPU's memory space. This is because the SMU views the entire physical

memory space as its own, which could lead to inadvertently overwriting portions of the PPU's memory space, causing such code to be corrupted if not careful. Additionally, the SMU must be prevented from executing instructions that reside within the PPU's memory space, as executing such instruction would pose a security risk to the SMU. As a result, the memory space must consist of regions with their own specific permissions. These regions apply only to the SMU's view of the memory space and are described below:

- PPU Shared Memory: This region maps directly to the PPU's physical memory space. It provides the SMU with read/write access to the PPU's memory for monitoring and data restoration tasks. This region is non-executable, as we want to prevent the possibility of the SMU inadvertently executing any malicious code that may reside within the PPU's memory space.
- Safe SMU-exclusive: This region acts as a safety measure. As we leave the decision of what data is stored by the monitoring hardware to the developer, we provide this region to store information that may be malicious in nature. An example would be storing a possibly corrupted block of instructions from the PPU's memory. Storing this in a non-executable memory space prevents the possibility of such code corrupting the OS/software executing on the SMU. This region can be non-contiguous, but must map to physical addresses that only the SMU has visibility into.

- Open SMU-exclusive: This region is the only region of the monitor’s memory space from which instructions can be executed. As such, it is used to store the monitoring OS and/or any software that executes on the SMU. This region can be non-contiguous, but must map to physical addresses that only the SMU has visibility into.

Memory regions that require instructions to be non-executable can be enforced through various hardware-based means. Processors based on a Harvard architecture naturally contain a non-executable memory space (the data space) as the data and instruction memories are separate. Processors supporting NX/XD bit technology described in 2.7.4 can also be used. If the processor is neither Harvard-based nor does it have native support for the no-execute bit, an execution policy enforcer like the one described in 3.5.2.1 can be implemented to perform a similar non-executable capability.

The asymmetrically shared main memory operates only on the physical memory space, as opposed to the virtual memory space. The physical memory space is the data stored in memory as seen purely from the hardware level. Virtual memory on the other hand organizes data into a number of pages that require the OS and page table directories (located in memory) to access. As this method relies on the SMU being able to address the PPU’s memory at the physical level, on its own, it can only enable security monitoring mechanisms that rely on physical memory introspection. CoPilot, for example, relies on such a capability [46]. However, this method can also

help to enable memory introspection into the virtual memory space as well when combined with the concepts proposed in Sections 3.5.4.2 and 3.5.4.3.

3.5.4.2 Co-opted Memory Management Unit. The CuPIDS prototype is able to use a separate processor to monitor the user space of code executing on a production processor by creating a monitoring shadow process that hooks into a production process' virtual memory space [59]. This is made possible by the single OS nature of CuPIDS which tightly couples the monitoring software to the production software being monitored. As our goal is to create a system where the monitoring software is as loosely coupled to the production software as possible, we have opted to execute completely separate software on each processor - preventing the creation of hooks into the virtual memory subsystem, as was done in the CuPIDS prototype. Therefore, we propose modifying the memory management unit (MMU) of the PPU in order to access state information associated with user-space processes executing on the PPU. It should be noted that as this method targets the virtual memory subsystem, this technique is intended for dynamic, multiprogrammed environments.

The MMU of a processor is responsible for controlling the translation of virtual memory addresses into physical memory addresses. In traditional computer architectures supporting virtual memory, the MMU resides within the processor itself. Therefore the MMU only services memory requests made by the processor core that contains it (i.e., contemporary multi-core processors have an MMU for each core). However, to monitor the state of user space processes in our system, we propose mod-

ifying the PPU’s MMU such that the SMU can take control of (co-opt) it. This enables the SMU to be able to access the virtual memory space of the currently executing process on the PPU. Additionally, it should be noted that we are not proposing modifications to the virtual memory system itself, but rather we are proposing to modify the way in which the MMU can be controlled, while still maintaining compatibility with the currently accepted virtual memory model. This allows us to continue to use the abstract concept of virtual memory, rather than having to worry about modifying the complex functionality of the virtual memory system itself.

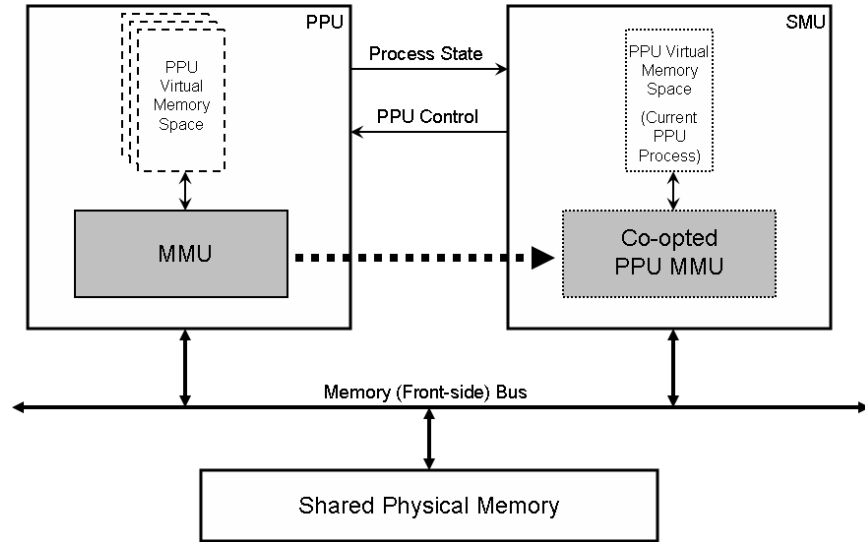


Figure 3.7: Co-opted Memory Management Unit High-level Architecture

The general architecture of the co-opted MMU concept is shown in Figure 3.7. It should be noted that we are not concerned with how the virtual address being monitored is obtained. Therefore, it is assumed that the SMU has knowledge of where key data structures to be monitored reside within a process’ virtual memory space. As the virtual address is known by the SMU, state information about the currently

executing process such as the PID, effective address, PC, etc. can be monitored via methods proposed in Section 3.5.1 and 3.5.2.1 to determine when the PPU accesses the data structure in question. The MMU is then co-opted and the physical address corresponding to the virtual address is retrieved and can be used by the SMU to access that portion of the PPU's virtual address space.

3.5.4.3 SMU with Multiple Memory Management Units. Another option for gaining visibility into the PPU's virtual memory space, is to use the MMU of the SMU itself, assuming the SMU includes an MMU. Contemporary processors contain an MMU and access a process' virtual memory space by updating a register used to store the value corresponding to the physical location of a process' page directory in memory. It is this register that determines to what virtual memory space addresses correspond as the processor executes code. As such, the processor can access the virtual memory space of any process within the system, so long as it is known where the page directory for a given process resides in memory. As the architecture can provide the SMU access to the PPU's memory space via asymmetrically shared main memory presented in Section 3.5.4.1, the MMU residing within the SMU has access to the memory locations containing the page directory for PPU processes. This enables the SMU to be able to access the PPU's virtual memory space, so long as it is provided the address where the particular page directory resides.

Using the SMU's sole MMU may introduce complications since the SMU will require access to its own code for execution, but is using its MMU to view the virtual

memory space of a process on the PPU. This can potentially result in the SMU not being able to return to executing its own code. To overcome such a complication, we propose modifying the SMU to contain multiple MMUs - a primary MMU and a secondary MMU. This is shown in Figure 3.8. The primary MMU is used only for memory accesses for code executing on the SMU itself. All accesses made by the SMU to the virtual memory space of a process executing on the PPU use the secondary MMU.

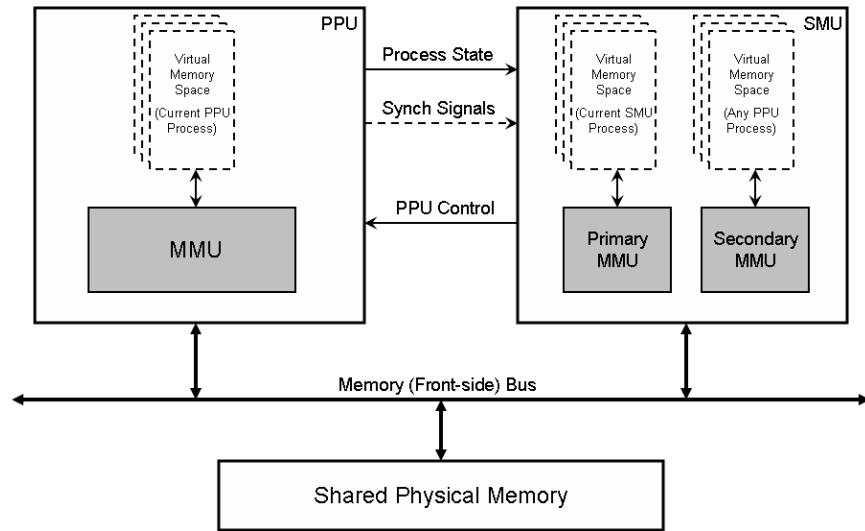


Figure 3.8: Multi-MMU SMU High-level Architecture

Monitoring the virtual memory space does not have to be limited to the currently executing process as was the case with MMU co-opting. This is a result of the SMU's secondary MMU being completely independent of the PPU, allowing the control register that points to the page directory to not point to the page directory of the process currently executing on the PPU.

In contemporary operating systems such as Windows and Linux, the scheduler decides when a process executes on the hardware. Processes that are not currently executing reside on the ready, I/O, or waiting queues [51]. At this point, the secondary MMU can be used to monitor the virtual memory space of the process waiting on one of these queues to perform a number of process integrity checks. We see performing such checks and other monitoring related tasks at this point as being able to provide five benefits:

1. **Ensured Trusted Execution:** Monitoring performed while the process is waiting on the ready queue can ensure that the process has not been compromised before it even executes on the PPU. When performed every time before a process is to be given execution time on the PPU, this can ensure that the process is always in a trustable state before it executes.
2. **Complex Algorithmic Monitoring:** While waiting on the ready queue, the state of the process is not changing. As such, the monitoring hardware does not have to keep pace with real-time execution. This allows the implementation of algorithmic monitoring operations that may not be feasible for real-time monitoring as the process executes.
3. **Efficient Resource Usage:** Not all processes within the system are necessarily monitored. Thus, the currently executing process may be one such non-monitored process, while a process on the ready queue can be a monitored one. In this case, the monitoring operations can be performed on the “ready” mon-

itored process at a time where the monitoring hardware would otherwise have gone unused.

4. Bad I/O Detection: Processes often times are waiting to receive data from some external source. As such a process is waiting, the state of the process can be recorded. After the process' I/O operation is complete and the process is waiting on the ready queue, the process could then be checked to determine if the I/O operation has damaged the process via a buffer overflow or some other form of input that may break the code. If detected, the recorded state gathered while the process resided on the I/O queue can be used to repair the damaged process.
5. Deadlock Detection: Synchronization mechanisms are implemented as a means to prevent multiple processes from accessing shared resources. However, in preventing simultaneous access to shared resources, the possibility of introducing a deadlock condition arises. While a process is waiting for a process to release a resource, it is placed on a waiting queue. Thus, using a second MMU in the SMU to check the state of processes as they wait on the waiting queue could be used to determine if a deadlock condition on the PPU currently exists.

It is also important to note that all of these capabilities apply to both kernel-level and user-level processes. Additionally, multiple monitored processes may be on the ready queue at any one time. For this reason, it may even be desired to include multiple secondary MMUs to be able to quickly and efficiently monitor multiple virtual memory spaces. This may be especially useful for item 5 presented above.

IV. Research Platform and Implementation

This chapter details the implementation of the architectural concepts proposed in Chapter III. To that end, we describe the hardware and software environments of the development platform upon which all implementations are constructed. We then discuss the details of implementing the various primitives on that platform. It should be noted that not all of the proposed primitives were able to be implemented using the development platform. In such cases, we discuss what is needed to implement these concepts, had the development platform allowed.

4.1 *Purpose of Implementation*

Implementation is done in order to demonstrate the functionality of the proposed primitives. In so doing, we help to validate our research hypothesis. The primitives are implemented in such a way as to adhere to the architecture proposed in chapter III. Although a number of the primitives are designed to be able to work simultaneously with other primitives, each primitive is implemented individually in order to show functionality as a proof of concept for that specific primitive. This is done for simplicity as well as to show the modularity of our architecture. For concepts that are not physically implemented, we describe what is required for such mechanisms to be implemented and argue the soundness of such concepts despite the lack of an actual implementation.

4.2 *Development Platform*

For prototyping purposes, the Xilinx ML310 development board, which is built around the Xilinx XC2VP30 Virtex-II Pro FPGA, is used. The XC2VP30 FPGA contains two embedded PowerPC 405fx cores and supports multiple instances of the Xilinx Microblaze softcore processor. The board itself contains a wide range of modern interconnects allowing the use of DDR SDRAM, a compact flash (CF) card reader, 10 BaseT ethernet, and USB 2.0 for JTAG debug information. More standard I/O interfaces such as serial (RS-232 UART), PS/2, and VGA are also present. It should be mentioned that we use a serial port exclusively for connecting to the computer where our development software resides. This allows commands and outputs to be sent via a hyperterminal communication interface. For further detailed information on the ML310 development board and the Virtex-II Pro FPGA, please refer to [65] and [67], respectively.

4.2.1 Embedded Processors. The Virtex-II Pro can implement processors based on both the IBM PowerPC and Xilinx Microblaze architectures. Both processor types vary in complexity/capability and are discussed below. The Leon3 softcore processor is also briefly described. Although the Leon3 processor is not a processor directly supported by the development platform, we felt it important to mention as it includes a number of capabilities that may be useful for future research efforts.

4.2.1.1 PowerPC 405fx. The Virtex-II Pro contains two IBM PowerPC 405fx (PPC405) hardcore processors integrated directly within the FPGA fabric.

As with all processors based on the PowerPC ISA, the PPC405 is a reduced instruction set computer (RISC) based on the von Neumann computing model. Although it is integrated within the FPGA fabric, the PPC405 within the Virtex-II Pro retains all of the same capabilities of its standalone counterparts. By today's embedded processing standards, the PPC405 is fairly sophisticated as it includes 16KB instruction and data caches, floating point logic, and an MMU to provide support for virtual memory and can operate at clock frequencies up to 400MHz. As such, the PPC405 can support multiprogrammed OSs such as Linux.

As two PPC405 cores are integrated into the XC2VP30, these processors can be used simultaneously and in tandem to complete tasks benefitting from multiple processors. However, the PPC405 does not implement cache coherency (i.e., the cache of each PPC405 core is completely independent of the other). According to a Xilinx engineer, a cache coherency mechanism may be able to be implemented, however, it would be very slow. For this reason, the PPC405 processors can not be used in an SMP fashion. This limits the PPC405 cores to only being able to execute different bodies of code simultaneously or to operate in lockstep when in a multiprocessor configuration. Such a configuration uses the two PowerPC processors to execute the same code simultaneously, where one PPC405 core updates memory and I/O and the other PPC405 core performs instruction/data integrity checks [43].

While the integrated PPC405 cores retain the same capabilities as their standalone versions, Xilinx has made some modifications to how the PPC405 cores interact with the FPGA fabric. Such modifications are in the form of wrappers that encompass

the PPC405 cores and are used to interface the PPC405 cores with Xilinx’s proprietary debugging hardware. These wrappers limit what pins are available to a system designer. As such, not all pins of the PPC405 are available for access by a system designer. Furthermore, the FPGA connects to the PPC405 cores only at the “pins” of the PPC405 cores, thus not allowing any further visibility into signals within the PPC405 core at any time other than when in a debugging mode. This limits the flexibility of the PPC405 cores. Additionally, in order to keep the number of pins to a minimum, certain signals, such as execution trace data, is output on a small number of pins and must be decoded. This makes the retrieval of such state information less than trivial. Due to such difficulties, the PPC405 is not well suited for our particular applications despite its impressive specifications. More detailed information on the PPC405 cores within the Virtex-II Pro FPGA can be found in [61, 66].

4.2.1.2 Xilinx Microblaze 5.0. The Xilinx Microblaze 5.0 is a softcore processor based on a Harvard architecture (i.e., separate instruction and data buses). As with the PPC405, the Microblaze is a RISC-based processor, however, it only contains a 5 stage pipeline. The softcore nature of the Microblaze allows it to be tailored to a specific application. As such, features such as cache, the inclusion of a floating point unit, and interrupt support are just a few of the features that can be configured. This makes for a very flexible platform for prototyping. Additionally, the Microblaze can even be augmented with application specific accelerators that tap directly into the execution stage of the pipeline via fast simplex link (FSL) connections. This can

greatly increase the performance of the Microblaze for a particular application. FSL connections can also be used to connect multiple Microblaze processors together in order to quickly share data.

Although the Microblaze is a simpler, albeit more flexible, design compared to the PPC405, it does provide a basic computational capability. This makes the Microblaze well suited to embedded systems and for basic prototyping. The Microblaze however, does not include an MMU of its own, so support for virtual memory is non-existent as a result. It should be mentioned, however, that as the Microblaze is a softcore processor, the processor has the potential to be modified to include an MMU. Extending this to having two or more MMUs could allow the Microblaze processor to act as part of an SMU that can perform virtual memory introspection via methods discussed in Section 3.5.4.3. Despite lacking native MMU support, a custom version of the Linux kernel, uCLinux, has been developed to run properly on the Microblaze.

As the Microblaze is a configurable softcore processor, I/O is not limited by a physical pin packaging as is the case with the PPC405. This allows for almost every signal within the Microblaze to be tapped into. Moreover, as headers that tap into certain signals are generated when the Microblaze processor is synthesized, there is no need to limit the number of I/O pins on the Microblaze, unlike with the PPC405 hardcores. This makes accessing processor state information much easier than the PPC405 hardcores. For these reasons, the Microblaze is flexible while still providing computational capabilities that suit our needs. As a result, the Microblaze softcore is

used for implementations requiring a processor. Please refer to [68] for more detailed information regarding the Xilinx Microblaze processor.

4.2.1.3 Leon3. The Leon3 is a softcore processor based on the Sun Microsystems SPARC V8 architecture. One notable feature of this processor is that it contains an MMU. As a number of concepts proposed in Chapter III rely on modifying the MMU functionality of a processor, the Leon3 softcore processor may allow for such modifications. The Leon3, however, is not compatible with Xilinx development tools and adheres to the AMBA bus standard, rather than the CoreConnect Bus Architecture used by the Microblaze and PPC405 processors. As a result, a completely different development environment would have to be used.

4.2.2 CoreConnect Bus Architecture. Xilinx FPGAs support the CoreConnect Bus architecture (CCBA). This feature allows the integration of the PPC405 and Microblaze cores within the XC2VP30 FPGA. The CCBA is based on a master/slave relationship with other system devices and supports three types of buses: the processor local bus (PLB), the on-chip peripheral bus (OPB), and the device control register(DCR) bus. The PLB is made for higher speed communication. As a result, the PPC405 processors can only connect to the PLB. Local Memory (dedicated solely to a PPC405 core) as well as a DDR SDRAM controller (available to all system devices) can also connect directly to the PLB to allow PPC405 processors high speed access to main memory. The OPB is slower compared to the PLB, and as a result is responsible for connecting the majority of system peripherals to any instantiated

PPC405 or Microblaze processors. Additionally, the OPB is the only bus to which Microblaze processors can directly connect. The OPB also supports a DDR SDRAM controller, for systems without a PPC405/PLB. The DCR allows configuration registers to be removed from a systems memory map in order to improve the bandwidth of the PLB. Figure 4.1 shows an example of how the various pieces of the CCBA fit together.

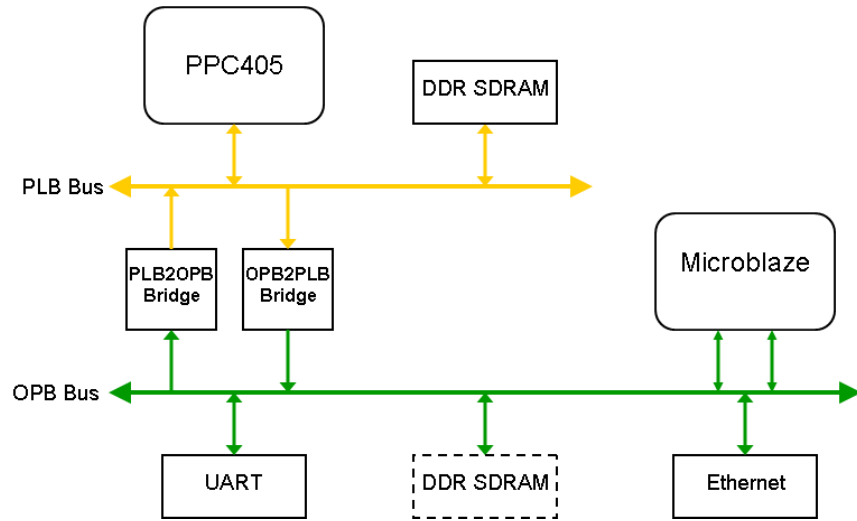


Figure 4.1: Example Embedded System Using Core Connect Bus Architecture

The PLB and OPB can be connected via PLB to OPB and OPB to PLB bridges. As a result, a PPC405-based system can be implemented to closely resemble the architecture of a standard personal computer, with the processor and memory residing on the front-side bus (i.e., the PLB) and the system peripherals residing on the back-side bus (i.e., the OPB). Additionally, the CCBA allows for multiple instances of these buses to be created within a single system. As a result, nontraditional computing architectures can be implemented to suit a particular application.

4.2.2.1 Xilinx IP Cores. Devices are instantiated on the PLB and OPB as IP Cores. IP Cores implement specific entities within the embedded system. Any device, whether it is a processor, memory, memory controller, I/O device, etc. can be instantiated within a design by adding its corresponding IP Core to the system. Most IP Cores are specifically designed to attach to the PLB and OPB within the system, so they are easily added to a design. Any processors that exist within the system communicate to the various IP Cores via memory mapped I/O. As an IP Core is seen as an addressable device within the system, a driver is needed for IP Cores residing on either the OPB or PLB to operate. This applies even if a dedicated OS is not explicitly loaded onto the embedded system. This is because any software loaded onto the embedded system is loaded in conjunction with the Standalone Board Support Package (BSP) unless an OS is specifically specified. The BSP is a set of modules that allows code to access the on-chip/on-board capabilities, such as caching and interrupts, in the absence of an OS [62].

Xilinx also includes a utility with their ISE Foundation known as CoreGen. The CoreGen utility allows an embedded system designer to quickly design an IP Core that can be used in an embedded system. IP Cores created with CoreGen can be designed to attach to the PLB or OPB. Cores that connect to a system bus through a specific controller, such as a memory block, can also be implemented. It should be noted however, that if creating an IP Core to connect to a system bus, a specific driver for that IP Core must be created as well.

4.2.3 Software Support. As the development board is based on a Xilinx FPGA, we are constrained to using mostly Xilinx development tools. For the design and construction of our embedded designs, the Xilinx ISE Foundation 8.2i and Embedded Development Kit (EDK) 8.2i design environments are used. These environments are described below. We also briefly touch on the embedded system debugging utility.

4.2.3.1 Xilinx ISE Foundation 8.2i. ISE Foundation 8.2i contains the tools required to successfully design and implement logic designs for Xilinx FPGAs. These tools are tied together using the Xilinx Project Navigator. The Project Navigator is an interface that provides for the creation and modification of logic designs using the Verilog and/or VHDL hardware description languages. Additionally, Project Navigator provides access to the entire Xilinx toolchain from design synthesis to downloading a generated bitstream to the FPGA.

As ISE Foundation includes the basic tools needed to design and implement logic in Xilinx FPGAs, the ISE is not intended to design and implement complete embedded systems. Rather, the ISE is specifically suited for designing and implementing custom logic designs. Designs requiring any kind of processing capability must use the Xilinx EDK package. However, it should be noted that ISE projects can contain embedded systems, however, these must be designed using the EDK and imported into the ISE project. As a number of our primitives use custom logic connected to the inputs and outputs of an embedded system, we extensively make use of this capability. Detailed information regarding ISE Foundation 8.2i is located in [60].

4.2.3.2 Xilinx EDK 8.2i. The Xilinx EDK 8.2i is an environment for leveraging the tools and resources of the ISE Foundation in a way that provides for the creation of entire embedded systems. As such, the ISE Foundation is required for the EDK to be able to operate. Through the EDK, the designer has access to the library of embedded processing cores and IPCore peripherals used for creating embedded systems on Xilinx FPGAs. The EDK is a self contained package, and as a result, a designer does not have to even directly use the ISE (via Project Navigator) if only creating embedded systems and nothing more. The EDK also includes access to the Xilinx synthesizer, allowing a system to be translated into a bitstream and downloaded to the FPGA without having to use the Project Navigator included with ISE Foundation.

As the EDK provides access to embedded system components, it also provides an environment for manipulating the parameters and connections of the system. The interface allows for the designer to control the number and type of processors, buses, and peripherals within the system and how they are all interconnected. Also provided is an interface to create, link, build, and debug software intended to execute on the embedded system. Additionally, the EDK provides access to Xilinx XMD, which is used to debug software as it executes on the embedded system. More detailed information regarding the Xilinx EDK platform can be found in [63].

4.2.3.3 Debugging Using Xilinx XMD. For debugging purposes, the XC2VP30 supports the Joint Test Action Group (JTAG) interface standard. As such,

Xilinx’s debugging shell known as XMD can connect to any JTAG enabled device within the system. This includes both the PPC405 and Microblaze processor cores, however, Microblaze-based designs require that a Microblaze Debug Module (MDM) IPCore be included in the system for XMD to be able to connect to the Microblaze processor. As XMD connects to a JTAG supporting device, XMD only operates once a bitstream has been used to configure the FPGA.

The XMD debugger is a command line driven interface that allows the designer to input commands in order to control the operation of any processor cores implemented in the current design. Standard debugging capabilities like stepping through code, reading and writing specific memory locations, etc. are present. For a list of XMD commands or details on how to use XMD, please refer to [63].

4.3 Linux Implementation

Linux is an open source operating system that has substantial industry support. Since our concepts are intended to be applied to actual operating environments, Linux was a natural choice for demonstrating the capabilities of our monitoring concepts. As a result, we have implemented a Linux-based operating environment on both the PPC405 and Microblaze processors. However, as complications with the Linux installations arose and design decisions were made, neither Linux installation is actually used in conjunction with any of the implemented functional primitives. Despite this, it is important to mention the work that has been done in regards to creating embed-

ded systems that can support an environment based on the Linux OS. Such efforts are described below.

4.3.1 Embedded Linux 2.4. As the PPC405 processor is popular in the embedded community, an open source version of embedded Linux based on the 2.4 kernel for the PPC405 processor is available. Using information presented in [8, 30, 42], we successfully implemented embedded Linux on a PPC405-based system. The tutorial detailing how this was done can be found in Appendix B.1. We were able to partition a compact flash card to contain the boot, swap, and root filesystem partitions. This provides access to a persistent storage medium similar to a hard drive. The embedded Linux installation was also able to utilize the ethernet port for network-based communications. Input and output was entirely console based via a standard terminal (i.e., serial port communication) interface. As a result, the Linux environment was very functional. However, for the reasons described in Section 4.2.1.1, the PPC405 is not used in any of our implementations. As a result, this embedded Linux environment is not implemented in conjunction with any of the implemented primitive

4.3.2 uCLinux. uCLinux is a version of Linux designed specifically to execute on processors lacking an MMU, and hence cannot utilize virtual memory. As a result, uCLinux is the only Linux-based kernel that supports the Xilinx Microblaze processor. Xilinx Provides the uCLinux sources files for compiling uCLinux for a Microblaze-based embedded system. Xilinx also provides documents detailing how

to set up the uCLinux cross-compiling environment and how to create a working uCLinux kernel in [41, 55].

uCLinux provides the same basic capabilities as embedded Linux on the PPC405, however there are still a number of differences. The most significant of which is that the Xilinx toolset did not provide the ability for mounting the root filesystem on the compact flash card. Instead, uCLinux was intended to mount its filesystem to a RAM disk created from the DDR SDRAM. As a result, all uCLinux kernel files resided in volatile memory. As certain implementations may have required slight modifications to the uCLinux kernel, this was not feasible since all changes to the kernel would be lost if power was ever taken away from the system. It should be mentioned that the uCLinux Kernel can access and use the compact flash as non-volatile storage, however, the bootloader provided with the uCLinux development environment did not support custom commands that would have allowed a root filesystem to be mounted on the CF card at system start up. As a result, a custom bootloader would have to be created to support this - something we did not have the experience nor the time to do. Due to this, uCLinux is not used any of the implementations.

4.4 Functional Primitives

The below sections detail the implementation of a number of primitives proposed in III. Some of the primitives were not implemented due to limitations inherent in current computer architectures. For the primitives that were not able to be imple-

mented, we discuss how we intend such designs to operate and what capabilities would be required to implement them.

4.4.1 Execution Policy Enforcement. The execution policy enforcement module, discussed in Section 3.5.2.1, is implemented using a Microblaze softcore processor as the PPU. The Microblaze connects to other peripherals within the system such as the RS232 UART (serial port), Microblaze Debug Module, etc. via an OPB. Rather than use DDR SDRAM as main memory, we opted to use 32KB of block random access memory (BRAM) with an address space ranging from 0x00000000 to 0x00000fff. We chose to use BRAM because programs executing from the DDR SDRAM must use a bootloader to actually load the program into memory and to start its execution. As BRAM is created from the FPGA fabric itself, the BRAM can be initialized when the bitstream configures the FPGA with our design, making implementation easier. The BRAM itself connects to a Local Memory Bus (LMB) and is dual ported, with one port connected to the data bus of the Microblaze and the other port connected to the instruction bus of the Microblaze. Additionally, the address space of both the instruction and data buses were made to overlap, so the data and instruction sides can both access the entire 32KB memory space.

As can be seen in Figure 4.2, the SMU is implemented using two modules - `noex_mem` and `noex_mem_en`. `Noex_mem` is a memory that correlates addresses in the PPU's memory space to whether or not those addresses are executable or not. `Noex_mem_en` is the enable logic for `noex_mem`. It ensures that the `noex_mem`

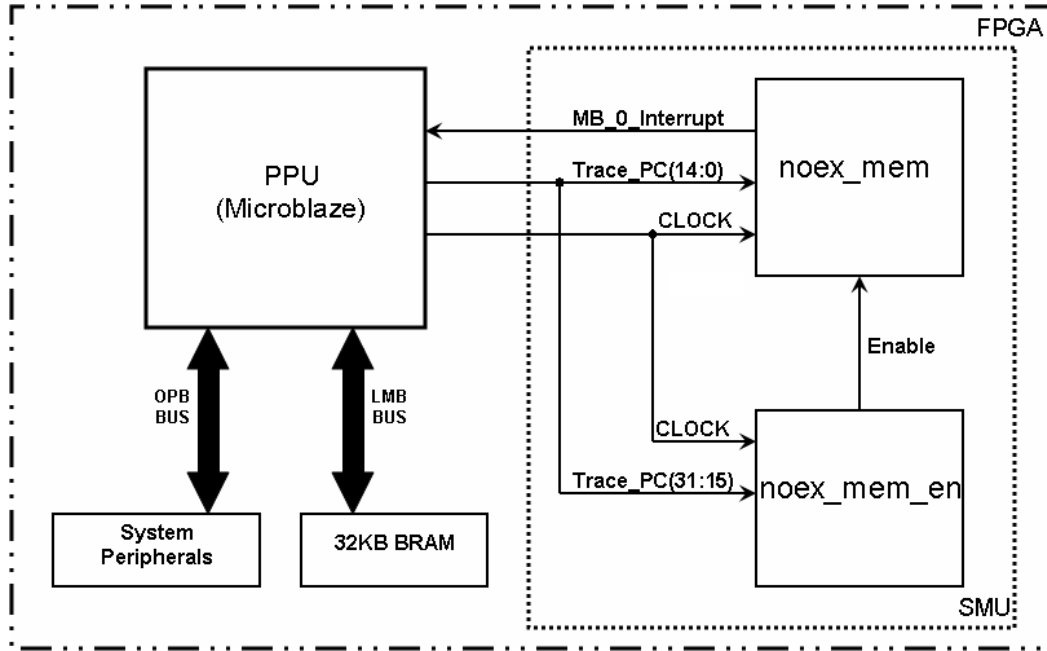


Figure 4.2: Program Counter Monitor Implementation

module is active only when the process it monitors is currently executing. Each module is constructed of RTL using the VHDL programming language. Please refer to Appendix A.1 for all VHDL code pertaining to the implementation of the execution policy monitor.

Noex_mem is an instance of BRAM containing 32768 memory locations. The number of memory locations is a limitation of the platform we are using, as the BRAM editor of Xilinx's CoreGen utility only allows the construction of a BRAM core with a maximum depth of 15 bits. As the memory space where our program will execute ranges from 0x00000000 to 0x00007fff, the PC of the executing code will only reside within that range. Therefore, the address range of the PPU's memory maps to the address range of the noex_mem module. As a result, we map the PPU's PC trace signals (Trace_PC) to the address inputs of the noex_mem module. Since the Trace_PC signal

consists of 32 bits, but the `noex_mem` module has only 15 address pins, only the 15 least significant bits (`Trace_PC(14:0)`) are connected to the `noex_mem` module. Each memory location is one bit wide, where “0” corresponds to an executable address and “1” corresponds to a non-executable address. The output memory output connects directly to the Microblaze’s external interrupt pin. As a result, if a non-executable memory location is executed, the `noex_mem` module will trigger an interrupt in the PPU.

The `noex_mem_en` module can be implemented to enable the `noex_mem` module in a number of ways. As this implementation is proof of concept, we chose the `noex_mem` module to always be enabled. As such, the `noex_mem_en` module is configured to output an enable signal when the 17 most significant bits of the PC (`Trace_PC(31:15)`) all equal “0”. This ensures that `noex_mem` is enabled only when PCs within the monitored address range are observed.

4.4.2 Multi-context Hardware Monitors. The multi-context monitoring concept relies on revealing the PID of the processes executing on the PPU at a particular point in time to the SMU. As mentioned in Section 3.5.1, the PID of each process is managed by the OS scheduler. For the current process, the scheduler points to a PID value stored in memory. This memory location resides within hardwired pages of the OS, meaning that the PID of the currently executing process always resides at a particular physical address. Consequently, that memory location can be monitored in order to retrieve the current PID and determine the currently executing process.

Rather than monitor memory itself by reading from a shared memory space, we tap directly into the main memory bus in order to retrieve the current PID. A schematic of our implemented multi-context hardware monitors concept is shown in Figure 4.3. It should be mentioned that while the implementation of our multi-context concept is displayed in terms of a structural block diagram, this is done purely for ease of explanation. The actual VHDL code written to implement the SMU in this case was written at the behavioral level. Connecting the SMU to the PPU was done at the structural VHDL level, however. Please see Appendix A.2 for all VHDL code related to this functional primitive.

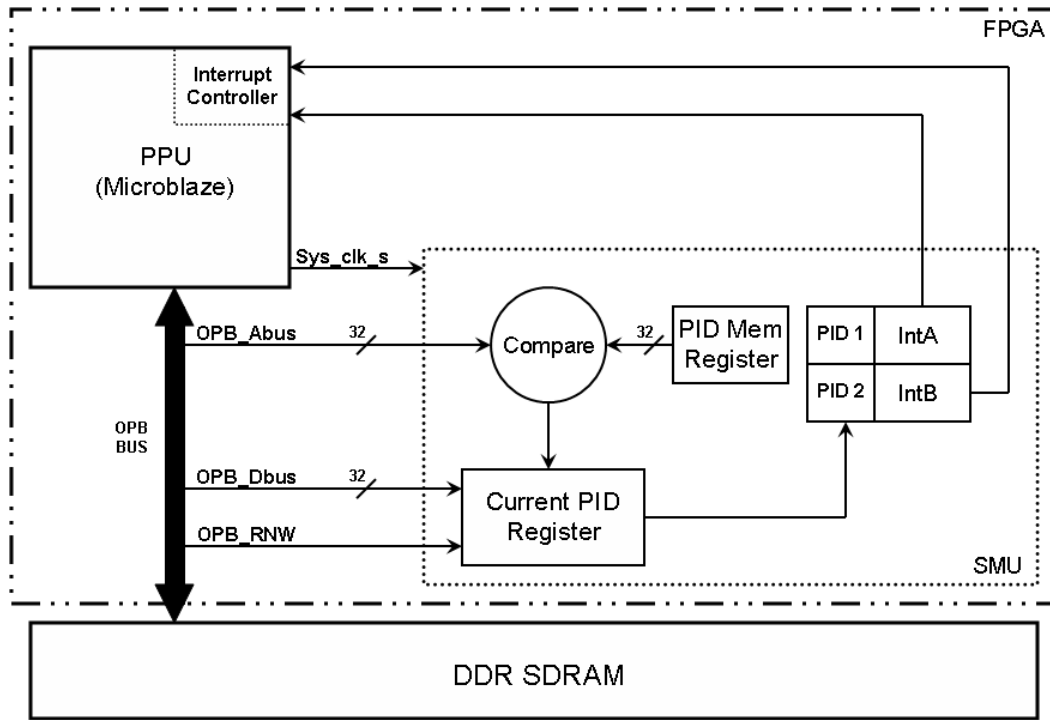


Figure 4.3: Logical Implementation of PID Retrieval

The PPU is implemented using a Xilinx Microblaze softcore processor with access to a 32KB local memory via an LMB, which is used to store the instructions

and data of the test program executing on the Microblaze. The PPU also has access to 128MB of DDR SDRAM for main memory. Access to memory occurs via an OPB, which acts as the main memory bus in this implementation. Other peripherals within the system such as the RS232 UART (serial port) and general purpose I/O are also connected to this OPB as well.

We tap directly into the main memory bus that the PPU is connected to in order to be able to view the PID of an executing process. As the PID capture logic requires access to the OPB to do this, the PID capture logic can be implemented as either an IPCore or as RTL. While implementation as an IPCore is most likely feasible to retrieve the PID from the OPB, we have opted to implement this particular primitive as RTL. We do this for a number of reasons: 1) ease of implementation, 2) guaranteed performance, and 3) to limit the PPU's visibility of the PID capture logic. Implementing the PID capture logic as an IPCore would have required the creation and use of a hardware driver. As tapping directly into a bus does not require a driver in this application, we opted to keep the implementation simple by using only RTL. Using RTL also makes it easier to ensure that the PID capture logic responds near instantaneously (within a single clock cycle) in a consistent fashion. Lastly, implementation as an IPCore makes the monitoring logic an addressable peripheral residing on the OPB. This would have made the PID capture logic visible to the PPU and would have required explicit communication with the PPU to even operate.

To retrieve the required information from the memory bus, the RTL taps directly into both the address (OPB_ABus) and data (OPB_DBus) lines of the OPB. We also

tap into the read-not-write (OPB_RNW) line, which is used to indicate whether a read or write operation is occurring on the OPB. The physical address corresponding to where the PID is stored is hard-coded into the RTL and is signified by the “PID Mem Register” (PMR) block in Figure 4.3. As we have visibility into the OPB_ABus signal, we compare it to the value stored in the PMR block to determine if the memory location containing the current PID is being accessed. When accessed, the compare logic will assert an enable signal to the Current PID Register (CPR). If the OPB_RNW signal is “0” (signifying a write operation) while the enable signal is asserted, the CPR latches the value currently on the OPB_DBus lines. This stored value corresponds to the PID of the newly executing process and will not change until another context switch occurs. The newly latched current PID value is then compared with values in a table containing PIDs corresponding to monitored processes. If the current PID matches one of the stored PIDs, an interrupt specific to that process is signaled. If the current PID does not match, no action is taken.

As mentioned in Section 3.5.1, there are three scenarios in which the PID is useful for monitoring, namely: monitoring a single process among many processes, monitoring many processes with a single monitor, and monitoring multiple processes with multiple monitors. Our specific implementation focuses on the latter case, as it is the most complex. As such, the table of PIDs corresponding to monitored processes contains two entries - the output of which is a single select signal for each PID. As this is a proof of concept, these select signals are not connected to actual monitors, but rather they are used as external interrupts connected the PPU. This allows us to verify

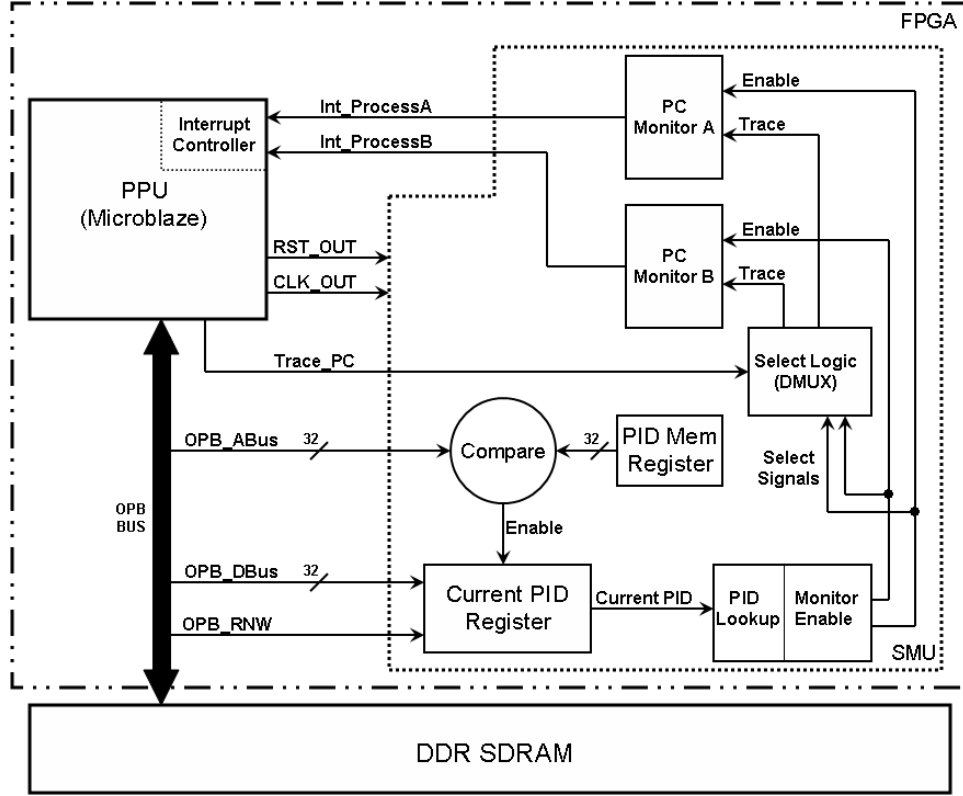


Figure 4.4: An Example of Monitoring Multiple Processes With Multiple Monitors

the operation of our PID capturing scheme, while not having to actually implement specific monitors. However, Figure 4.4 does provide an example of an SMU with multiple monitors monitoring multiple processes. Furthermore, as the Microblaze only supports a single interrupt natively, an interrupt controller is required for the Microblaze to be able to discern between the two interrupts.

4.4.3 Peripheral Access Control. The peripheral access control concept presented in Section 3.5.3 was not implemented on the development platform, however, we did generate a design that we believe can implement such a monitor. The proposed peripheral access control monitor is depicted in Figure 4.5.

In order to determine if a process has the rights to access a particular peripheral, two tables are utilized: the Peripheral Access Table (PAT) and the Peripheral Map Table (PMT). The PAT stores the PID of all monitored processes and correlates them to a peripheral access code that represents what peripherals within the system the process is permitted to access. The PMT associates a range of addresses (corresponding to the memory map of the peripherals) to a logic vector (i.e., the peripheral code). Figure 4.6 depicts an example of both the PAT and PMT. The peripheral access code is shown as having eight bits, where each bit represents a device in the system. As a result, eight peripherals would be supported in this system. A value of “1” denotes that the process is allowed to access the corresponding peripheral and a “0” denotes no access.

When a process accesses a particular memory address, the PAT and the PMT output the peripheral access code and the peripheral code, respectively. A logical AND function is used to compare these two values in order to determine if the current process is allowed to access the peripheral. If access is allowed, at least one of the outputs of the AND gate will be “1”. An logical OR function is then used to reduce the outputs of the AND gate to a single value which is then inverted. If the process is not allowed to access the particular address, the SMU will trigger an interrupt on the PPU. Additionally, it should be mentioned that as a process may access addresses not corresponding to a system peripheral, such addresses must be listed in the PMT with a corresponding peripheral code containing all “1”s. As a result, such addresses

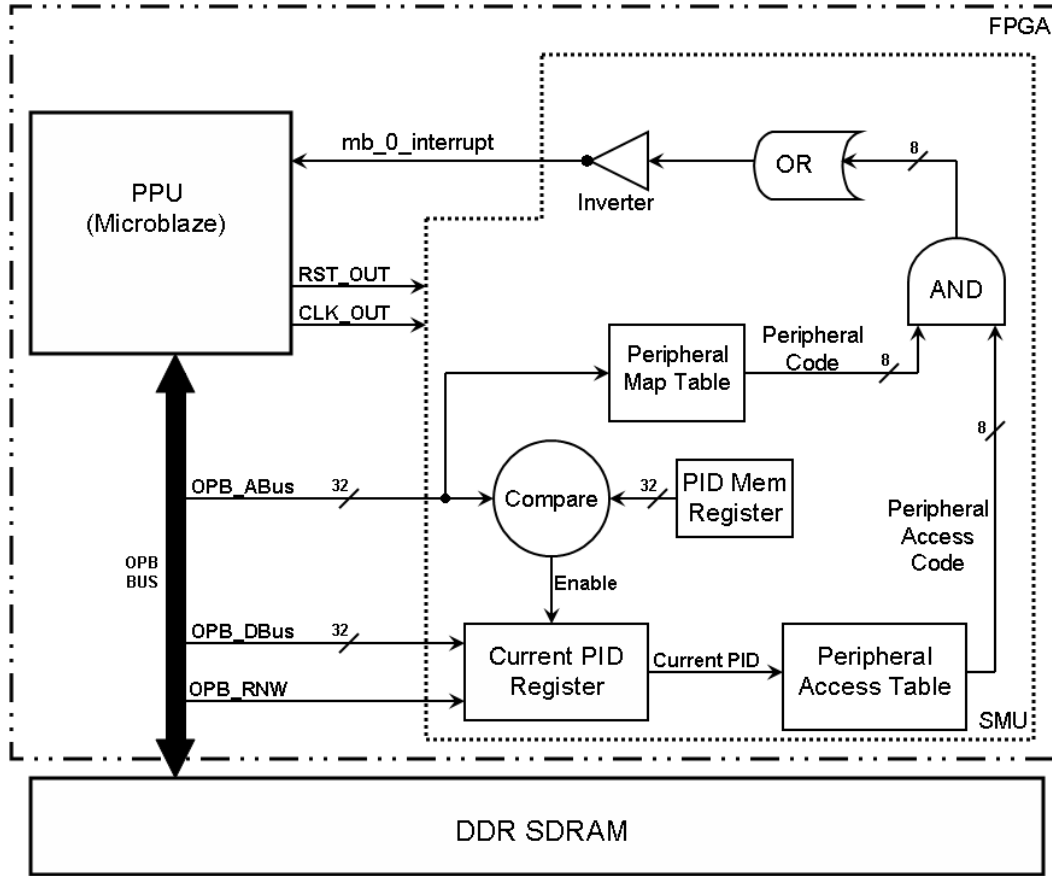


Figure 4.5: Planned Peripheral Access Control Implementation

will cause the output of the AND gate to always contain a “1”, thus ensuring that the SMU will not trigger an interrupt for such memory accesses.

For example, if the process corresponding to PID #1 of the PAT in Figure 4.6 attempts to access a peripheral at the address range 0x0010-0x001f, the operation would be allowed to continue since performing a logical AND of the access code (01000000) with the peripheral code (01000000) results in at least one bit value being “1”. However, if the process with a PID of “1” tried to access the peripheral at 0x0000-0x000f, the operation would not be allowed as the resulting AND would produce all “0”s. Additionally, the process corresponding to PID #8 would be allowed to access

Peripheral Access Table		Peripheral Map Table	
PID	Access Code	Address Range	Peripheral Code
1	010000000	0x0000-0x000f	10000000
8	11111111	0x0010-0x001f	01000000
⋮	⋮	⋮	⋮
5	00000001	0x0070-0x007f	00000001
10	10011010	> 0x007f	11111111

Figure 4.6: PAT and PMT Example

all peripherals within the system as a logical AND of its access code and any peripheral code will always result in at least one bit value being “1”.

4.4.4 Asymmetrically Shared Main Memory. To realize the asymmetrically shared memory concept proposed in Section 3.5.4.1, we implement both the PPU and SMU using Microblaze softcore processors. This is done since the Microblaze can easily access memory. 128MB of DDR SDRAM is used as the shared memory. Access to the shared memory is provided by a multi-port memory controller described below.

4.4.4.1 Multi-Port Memory Controller. A Microblaze processor typically accesses the DDR SDRAM by physically residing on the same OPB as the DDR SDRAM controller. If multiple processors require access to memory, as in our case, both processors would reside on the OPB as the DDR SDRAM controller. However, due to how the DDR SDRAM controller functions, there is no way to allow different processors to have access to different regions of memory. Moreover, even if the DDR SDRAM controller did allow this, there is still the issue of creating regions of memory with varying privilege levels. As a result, the default DDR SDRAM controller that

Xilinx provides with its development suite cannot be used in this implementation. Instead, a multi-port memory controller developed by Xilinx is used. This multi-port memory controller can provide multiple processors access to varying memory regions.

The multi-port memory controller we use is the Multi-Port Memory Controller 2 (MPMC2) developed by Xilinx. It was originally designed for applications requiring high performance access to memory in multiprocessor embedded applications. Although our application does not necessarily require high performance access to memory, it does allow us to connect multiple processors to a DDR SDRAM module and control the region of memory that each processor can access. A diagram of the MPMC2 is shown in Figure 4.7 below.

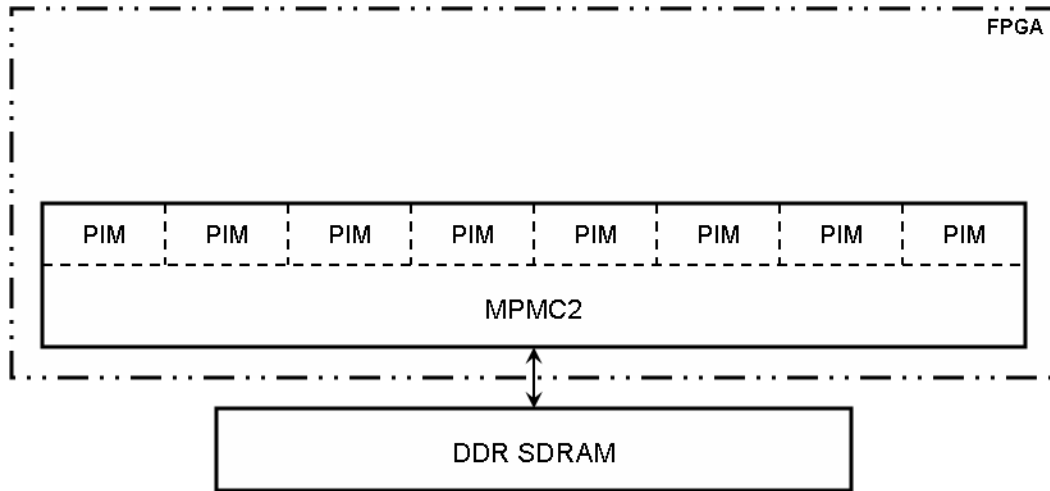


Figure 4.7: MPMC2 Basic Organization

The MPMC2 connects to devices within the embedded system through a number of Port Interface Modules (PIM). Up to 8 PIMs can be utilized at any one time, and each PIM supports connections to all buses supported by the CoreConnect Bus architecture, as well as the Xilinx Cache Link (XCL) and Communication Direct

Memory Access Controller (CDMAC) interfaces. Furthermore, custom made devices that do not reside on a CoreConnect supported bus can be connected directly to a PIM using the Native Port Interface (NPI). For more information regarding the design of the MPMC2 and how to configure a system utilizing it, please refer to [69–71].

4.4.4.2 System Construction. While certain regions of our asymmetrically shared main memory concept can apply to a non-contiguous memory space, as was pointed out in Section 3.5.4.1, for the ease of prototyping, the asymmetrically shared memory is implemented using contiguous regions. A logical view of the memory space as seen by both the PPU and the SMU is displayed in Figure 4.8 (not drawn to scale). In order to make the SMU’s memory space invisible to the PPU (keeping the SMU as secure as possible), the software on the PPU is limited to addressing only a 64MB portion of the 128MB of available physical memory space. This is accomplished by configuring the PIM connecting the PPU to the MPMC2 to only be able to access memory ranging from address 0x00000000 to address 0x03ffffff. The SMU attains visibility into the PPU’s memory space by configuring the PIMs connecting the SMU to the MPMC2 to have access to the entire physical memory space (i.e., ranging from address 0x00000000 to address 0x07ffffff). As such, the SMU has access to its own contiguous memory space while still being able to access the PPU’s memory space for security related monitoring. While this serves to distinguish what regions of memory are visible to the PPU and the SMU, it does not fully provide for the varying permission levels associated with the different regions.

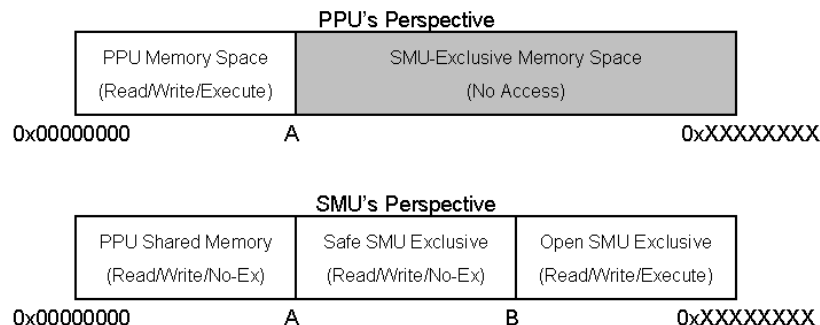


Figure 4.8: Memory Map and Permissions as Viewed by the PPU and the SMU

The regions of varying permissions are dependent on how the PPU and SMU are physically connected to the MPMC2. Figure 4.9 depicts the physical architecture of our asymmetrically shared memory system. As mentioned previously, the PIM that connects the PPU's OPB is configured to only allow access to addresses ranging from 0x00000000 to 0x03ffffff. This is done by connecting both the data and instruction buses of the PPU to the MPMC2 via a single OPB. In so doing, the PPU has read, write, and execute privileges for its memory space.

The Harvard architecture of the Microblaze processor is leveraged to create the different regions of varying permissions that the SMU has access to. As a Harvard architecture has separate data and instruction buses and memories, any addresses associated with the data side can only be used to read and write data, while addresses associated with the instruction side can only be used to read instructions (i.e., such memory locations are executable). Thus, the permissions of varying regions of memory are controlled by mapping the instruction and data buses of the SMU to different regions of memory. This is accomplished by mapping the data and instruction buses of the SMU to different OPBs. This allows the instruction and data buses of the SMU

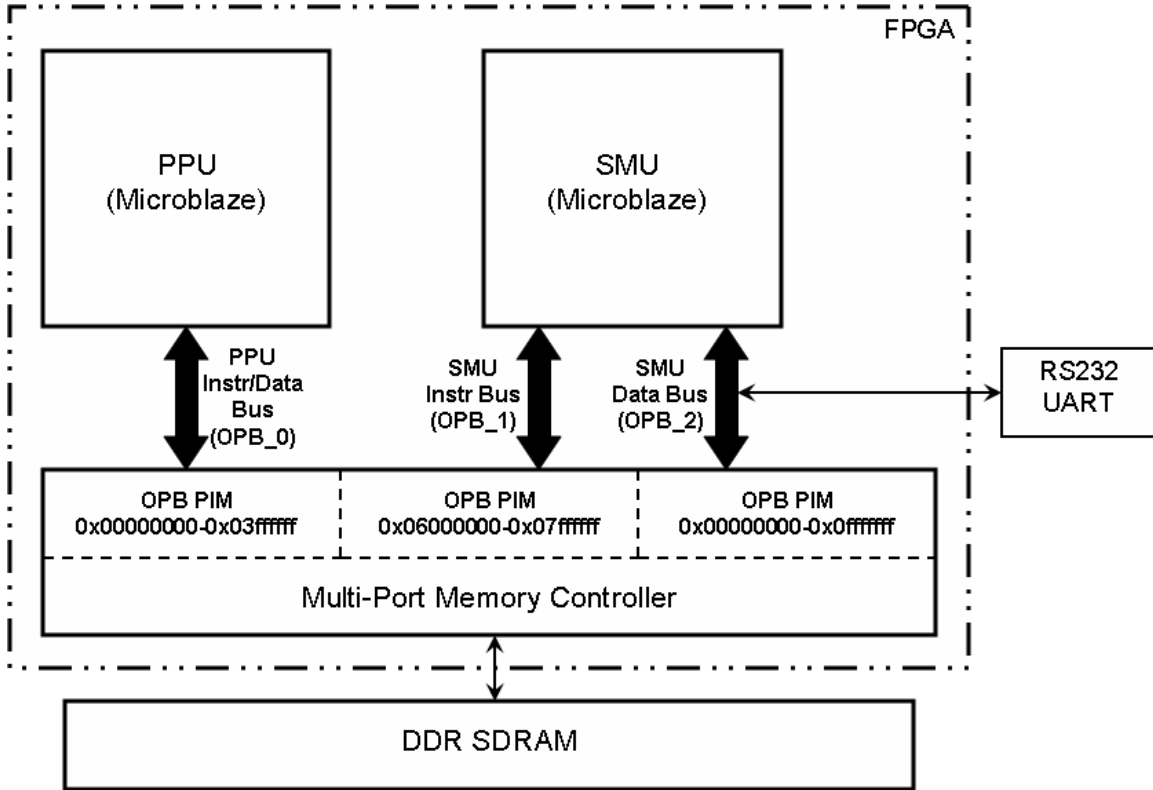


Figure 4.9: Asymmetrically Shared Memory Implementation

to be connected to separate PIMs - each with a different address range. As such, the SMU's data-side PIM is configured to allow access to a 256MB address space. The first addressable 128MB ranges from 0x00000000 to 0x07ffffff and, as a result, the SMU has read/write access to the entire shared memory. The second 128MB ranges from 0x08000000 to 0x0ffffff and allows for peripherals to connect to the SMU's data-side OPB. Additionally, the SMU's instruction-side PIM is configured to allow access to a 32MB portion of the shared memory ranging from addresses 0x06000000 to 0x07ffffff. As a result, this region of memory allows the SMU to not only read and write data, but to read instructions as well, making this region of memory executable.

By configuring our system in this manner, we produce the asymmetrically shared memory space depicted in Figure 4.8.

4.4.4.3 Application Example - Enhanced CoPilot System. Without the aid of other primitives, our asymmetrically shared memory system allows the SMU to have direct access into the physical memory space of the PPU. As a result, this system can be used to implement the same capabilities as the CoPilot system that was discussed in Section 2.6.2.2. Whereas CoPilot uses an add-in-card (with its own dedicated memory) and monitors the production processor's memory space via a PCI bus, our implementation sacrifices some of the memory available to the PPU in order to allow the monitoring coprocessor to reside in the same physical chip package as the PPU. As is the case with CoPilot, by itself, our implementation would still be limited to only being able to monitor the pages that are hardwired into memory. However, this monitoring capability can be extended to the virtual memory space by also implementing our MMU co-opting or multiple MMU concepts presented in Sections 4.4.5 and 4.4.6, respectively. Additionally, as the SMU resides at the same logical level as the PPU, the SMU has the ability to exert control over the PPU. As a result, using the asymmetrically shared main memory in the manner proposed here can remedy one of the largest shortcomings of the CoPilot system. Additionally, the asymmetrically shared main memory allows the SMU to have access to main memory without having to contend with other system peripherals - the constraining

factor causing CoPilot to perform monitoring only every 30 seconds. As a result, our approach could provide monitoring at a faster rate than that of the CoPilot system.

4.4.5 Co-opted Memory Management Unit. As we do not have access to a softcore processor containing an MMU, the MMU co-opting concept, presented in Section 3.5.4.2, cannot actually be implemented at this time. Rather, we discuss what we believe will be required in order to provide such a capability.

4.4.5.1 Hardware Support. The SMU gains visibility into the memory space of the currently executing process by querying the PPU’s MMU to translate a virtual address provided by the SMU. This will result in the retrieval of the data at the specified virtual address. The architecture we propose for doing this is shown in Figure 4.10.

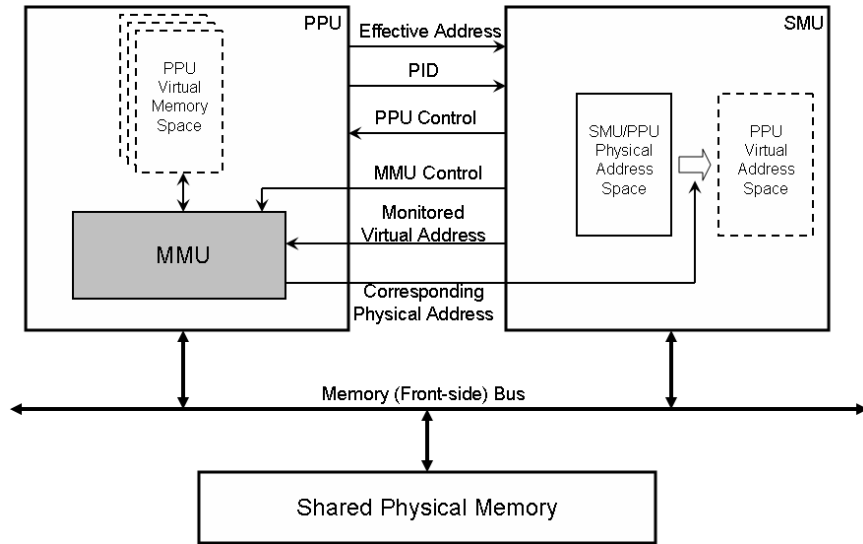


Figure 4.10: Proposed Co-opted Memory Management Unit Architecture

In order for the SMU to be able to determine what process is accessing memory and where, two pieces of state information are needed - the PID and the effective address. The PID indicates what process the PPU is currently executing, allowing the SMU to determine if the code currently executing corresponds to a monitored process. The PID can be gathered via methods presented in 4.4.2. The effective address corresponds to the address within the current virtual memory space, which is used to indicate what virtual memory address is currently being accessed by the PPU. Gaining insight into both of these pieces of state information can give the SMU visibility into how the PPU is executing a particular process, however it does not provide the SMU itself with visibility into the current virtual memory space.

Visibility into the virtual memory space of the currently executing process is provided by co-opting the PPU's MMU. When a monitored process accesses a virtual address containing a key data structure, a control signal is first sent from the SMU to the PPU notifying the PPU that its MMU is about to be co-opted. At this point, any memory operations currently in progress are allowed to complete so as to not corrupt any data. As the SMU will be sending a virtual address to the PPU's MMU and an address (in the form of data) in return, the SMU must have direct connections from its address and data lines to the PPU's MMU. As a result, when the MMU is co-opted by the SMU, the address and data lines of the SMU connect to the PPU's MMU, rather than to the memory bus. Additionally, at this time the PPU will have no access to the MMU. As a result, either the entire PPU must be halted or the PPU can continue execution so long as no memory access instructions are executed. It

should be noted that this will most likely require a change to the control logic of the PPU as the MMU may not function if the rest of the PPU is halted.

Now that the SMU has exclusive access to the PPU's MMU, the SMU can send the monitored virtual address to the MMU. As the MMU is still linked to the virtual memory space of the currently executing process, the MMU will translate the virtual address to a corresponding physical address which is then transmitted to the SMU. It should be noted, however, that the PPU's MMU must be modified to be able to return the physical address, as the MMU usually performs translation for specific memory operations. Once the SMU has received the corresponding physical address, the SMU sends a control signal that relinquishes the SMU's control of the MMU and resumes the PPU's execution of the process.

Once the SMU has retrieved the physical address of the data structure in question, the physical address can be used by the SMU to access the desired data and any necessary checks can be performed. This assumes that the SMU has visibility onto the PPU's physical memory space. This can be done using a method like the asymmetrically partitioned main memory concept described in 3.5.4.1.

4.4.5.2 Software Support. When the SMU co-opts the PPU's MMU, the PPU can either halt entirely, or execute instructions until a memory access instruction is encountered. As this is controlled in hardware by the SMU, the code being executed on the PPU should have no awareness of the MMU's ability to be co-opted. Thus, no changes should be required to the software/OS executing on the

PPU to enable such a capability. However, as the the SMU needs to be able to access a device (i.e., the MMU) in a particular way, this will most likely be implemented by a new instruction made specifically for co-opting another processor's PPU. As a result, the OS/software executing on the SMU must be aware of such a capability, but the changes required to do so should not be very extensive.

4.4.6 SMU with Multiple MMUs. The multiple MMU concept, presented in Section 3.5.4.3, cannot actually be implemented at this time. This is due to the architectural limitations of the processors available to us. As a result, we discuss what we believe will be required in order to provide visibility into the virtual memory space of monitored PPU processes by incorporating multiple MMUs into the SMU.

4.4.6.1 Hardware Support. In Intel's IA-32 architecture, an MMU accesses the virtual memory space of a particular process by updating Control Register 3 (CR3) with a physical address that points to the page directory of a particular process [25]. This allows any memory accesses by the currently executing process to have access to its virtual memory space. The CR3 register is updated implicitly when the OS scheduler performs a context switch to begin execution of another process, however, it can also be explicitly updated via an instruction integrated in executing code [25]. As a result, we can specifically control what virtual memory space the MMU accesses to perform virtual address translation so long as the address corresponding to the desired page directory is known. We believe this capability can be leveraged in such a way as to allow the SMU to have access into the virtual memory space of the

PPU. In order to do so, however, a number of architectural changes need to be made at the processor level.

Unlike the MMU co-opting concept detailed in 4.4.5, our multiple MMU concept provides the SMU visibility into the PPU's virtual memory space by using the SMU's own hardware resources rather than tapping into the hardware resources of the PPU. As a result, the architectural changes are primarily targeted at the processor being used to implement the SMU. The most blatant of these changes is the addition of at least another MMU to the SMU. Thus, the SMU contains a single primary MMU and one or more secondary MMUs. The primary MMU is used for servicing memory accesses made by the monitoring code executing on the SMU itself. As such, the primary MMU fetches instructions and data to execute code executing on the SMU, thereby fulfilling the same role as an MMU in any other processor supporting virtual memory. The secondary MMU(s) are only used to gain access to the virtual memory space of PPU processes, therefore, the secondary MMU(s) do not fetch executable instructions. This helps to prevent the SMU from being compromised as potential malicious instructions gathered from the PPU's memory space cannot be executed by the SMU.

As the primary MMU and secondary MMU(s) perform different roles within the SMU, added instruction support is required. The obvious instructions to be added are load and store instructions that specifically leverage the secondary MMU(s). For this to occur, additional load-store memory access units must be added to the SMU's

pipeline. These additional load-store units only operate on instructions that leverage the secondary MMU(s).

Although multiple MMUs are used to access the PPU's virtual memory space, virtual memory maps to physical memory. Therefore, it goes without saying that the SMU must have access to the PPU's physical memory space. As a result, the SMU and the PPU must share memory. For the protection of the SMU itself, the shared memory should adhere to the model discussed in 3.5.4.1. This provides the SMU access to the PPU's memory space, while keeping the SMU's memory space invisible to the PPU. As mentioned previously, since a secondary MMU cannot fetch instructions for the SMU, the SMU is physically protected from executing possibly malicious code that may be retrieved from the PPU's memory space as a result of monitoring. This provides for non-executable regions of the asymmetrically shared memory model to be implemented.

While we have described the hardware-level changes that must be made in order to gain visibility into the PPU's virtual memory space via our multiple MMU concept, we have not discussed the hardware support that is required for the SMU to know when to leverage that visibility. The SMU (invisibly) gathers PID and effective address state information from the PPU via hardware-based methods discussed in 4.4.2. This state information is used by the SMU to determine when a particular process is executing on the PPU and when that process accesses specific virtual addresses. It should also be mentioned that, as the page directory address is different for every process, there is a 1:1 mapping between the PID and the page directory address. As a

result, a page directory address can be used to determine the identity of the currently executing process. This assumes that the SMU has knowledge of the particular page directory address in question as well as knowledge of what process it corresponds to. For the SMU to acquire information regarding a process' page directory, software support is needed. Software changes are also required in order for the SMU to be able to monitor currently non-executing processes. These software-level changes are discussed below.

4.4.6.2 Software Support. While the multiple MMU concept provides the SMU visibility into the PPU's virtual memory space, this is dependent on the SMU having the address of the page directory corresponding to the desired virtual memory space. As each process has a unique virtual memory space, each process' page directory address is unique. This address can be explicitly communicated to the PPU when a monitored process is created. This communication should only be handled by a kernel level process on the PPU. Furthermore, the kernel should be in a known trusted state, so it can be assured that the value has not been altered for malicious purposes. This is similar to the creation of a monitored process in CuPIDs [59].

Although multiple MMUs can provide for monitoring processes that are not currently executing, the SMU must have information regarding whether the process(es) in question are on either the ready, I/O, or waiting queues. The concept of queues, however, is a software-level construct that is controlled by the OS, and as a result, cannot be determined at the hardware level. Thus, changes to the OS executing on

the PPU are required to notify the SMU when a particular process goes on a certain queue. This requires some explicit communication from the PPU to the SMU. As the OS scheduler coordinates the processes on the queues, it is most likely that the scheduler itself would control such communication as it schedules processes for execution. It should also be noted that although the ability to monitor processes not currently executing relies on software-level changes, for security purposes, dedicated logic connecting the PPU and the SMU should be used, rather than communicating this over the front-side bus.

Changes pertaining to the SMU's OS/software that help to enable the multiple MMU concept are minimal compared to the changes required for the PPU's OS/-software. As mentioned previously, new instructions are required for the SMU to be able to leverage the secondary MMU(s). Although these instructions are physically implemented as logic in the SMU, the OS/software executing on the SMU must be aware of these new instructions in order to take advantage of them. As a result, the software executing on the SMU needs to be coded with these instructions in mind.

V. Testing and Results

In this chapter we present the testing methodology and results of the implemented primitives. As the proposed functional primitives are implemented as proof of concept, simulation and testing are primarily functional in nature. For this reason, we integrate the testing methodology, any simulations, and implementation results into a single chapter. It should be noted, however, that a number of the primitives were not implemented and/or tested. In such cases, we explain the reasons that attributed to this and comment on the expected results.

5.1 *Execution Policy Enforcement Module*

The execution policy enforcement module was successfully implemented on the development platform. As such, logic simulation and actual testing was able to be performed, the results of which are described below.

5.1.1 Testing Methodology. Testing for this primitive is functional in order to show proof of concept. As described in Section 4.4.1, the execution policy enforcement module monitors the PC of the code currently executing on a Microblaze processor. Monitoring is performed by the non-executable memory module, the output of which acts as an interrupt for the Microblaze processor. As a result, there are a number of cases that we test to ensure proper operation of execution policy enforcement logic. These test cases are described below.

1. Executable Instruction: This case tests the output of the of the execution policy enforcement module when an executable instruction executes. If implemented correctly, the interrupt will not be triggered.
2. Executable Instruction Follows an Executable Instruction: This case tests the output of the execution policy enforcement module when an executable instruction executes after an executable instruction. If implemented correctly, the interrupt will not be triggered.
3. Non-Executable Instruction Follows an Executable Instruction: This case tests the output of the execution policy enforcement module when a non-executable instruction executes after an executable instruction. If implemented correctly, the interrupt will be triggered.
4. Non-executable Instruction Follows a Non-executable instruction: This case tests the output of the execution policy enforcement module when a non-executable instruction executes after a non-executable instruction. If implemented correctly, the interrupt will continue to be triggered.
5. Executable Instruction Follows a Non-executable Instruction: This case tests the output of the execution policy enforcement module when an executable instruction executes after a non-executable instruction. If implemented correctly, the interrupt will no longer be triggered.

5.1.2 Simulation. Simulation of the execution policy enforcement module was performed only for the logic implementing the monitoring mechanism. The em-

bedded system connected to this logic in the implementation was not part of the simulation, since the functionality of the execution policy enforcement module can be tested without the use of the embedded system. As such, the logic has two inputs - *clock* and *pc_bits*, and one output - *mb_int*. The *clock* signal is self explanatory. The *pc_bits* signal represents the program counter trace that is output from the Microblaze processor during execution. The *mb_int* signal is the interrupt signal used to signal when a non-executable instruction has been reached. The waveforms corresponding to these signals can be found in Figure 5.1 below.

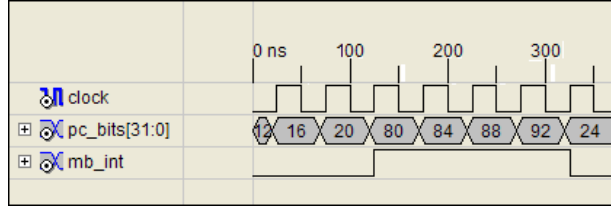


Figure 5.1: Execution Policy Enforcement Simulation Result

The simulation uses a 50ns clock cycle. The *pc_bits* test waveform input is made to resemble a malicious event. This is done by the waveform incrementing the *pc_bits* value as if it were executing sequential code that suddenly jumps to a malicious region of code. This is represented by *pc_bits* changing from 20 to 80 in the simulation waveform above. The “malicious code” executes by the *pc_bits* waveform incrementing from 80 to 92. The *pc_bits* value then changes to 24, representing a return to the valid stream of instructions.

The non-executable memory module (*noex_mem*) is initialized for all program counter values 80 and above to be non-executable. As a result, any PC value of 80 and above should set the *mb_int* signal. As can be seen in Figure 5.1, this is confirmed

as *mb_int* raises from “0” to “1” when *pc_bits* becomes 80. Subsequent malicious instructions (i.e., *pc_bits* ranges from 84 to 92) continue to keep *mb_int* set. The *mb_int* signal then returns to “0” when the “malicious code” returns to the original stream of instructions via the *pc_bits* value being 24. It is also important to note that the simulation shows that the execution policy enforcement module can respond in under one clock cycle of receiving a PC value.

5.1.3 Implementation Results. Similar to the simulation above, we created code to test the the functionality of the execution policy enforcement module. The code we used is as follows:

```
//interrupt service routine//

void noex_int_handler(void *arg) {

    print("No-Execute Memory Location Reached!");
}

int main (void) {

    microblaze_enable_interrupts();

    print("-- Entering main() --\r\n");

    print("-- About to jump to non-executable memory location--\r\n");

    __asm__("bri 0x0400");

}

print("-- Exiting main() --\r\n");
```

```
    return 0;

}
```

As can be seen, the code consists of two “print” statements, followed by a “bri” inline assembly instruction, followed by a final “print” statement. The code is simple, but is intended to work in the same way as the simulation’s *pc_bits* waveform. The “bri” instruction is an unconditional branch to the specified address. Thus, as soon as this instruction is reached the PC should branch to address 0x0400. We set the jump to this address because this memory location resided just outside of the code’s text segment. This ensures that the addresses that we deem as “non-executable” are not real instructions that are supposed to be executed by the code. Additionally, the `noex_mem` VHDL module was configured to mark addresses 0x0390 to 0x0500 as non-executable. Thus, when the unconditional branch updates the PC to 0x0400, the `noex_mem` module should output an interrupt, which the Microblaze receives, signalling the Microblaze to execute the code located in the interrupt service routine at the top of the code segment.

Upon executing this body of code, however, the hyperterminal (used to display the output) displayed the first two “print” statements in the main body of the code over and over in an infinite loop. Other addresses outside of the text segment were attempted, but the same result was observed. This is most likely being caused by an exception within the Xilinx standalone BSP that gets loaded to the FPGA along with this code. We also attempted to jump to addresses within the code’s text segment, however, this resulted in crashing the program. As a last resort, we modified

the configuration of the `noex_mem` module to mark the entire text segment as non-executable so as to make sure instructions being executed would trigger an interrupt. Re-executing the code resulted in the interrupt service routine being executed in an infinite loop. Although it was shown to work under a less than optimal test case, we were able to show that the implemented execution policy enforcement module functions properly.

5.2 *Multi-context Hardware Monitors*

The multi-context monitoring concept was able to be implemented on the development platform. As such, simulation was performed, however, actual testing was not able to be conducted. In this section, we describe our testing methodology, simulation results, and the issues encountered that prevented us from performing actual functional testing.

5.2.1 Testing Methodology. Testing for this primitive is functional in order to show proof of concept. As described in Section 4.4.2, the multi-context monitoring design monitors a specific memory address for specific PID values in order to determine the currently executing process. As not all attacks change the control flow of the targeted program, the address where the PID value resides and the values of the monitored PIDs themselves are all arbitrary in our testing. Similar to how execution policy enforcement was tested in Section 5.1, interrupts connected to the Microblaze processor (i.e., the PPU) are used to notify when a PID value matches that of a monitored process.

Testing consists of a number of memory accesses that read and write varying data values to a number of memory addresses to demonstrate the functionality of the multi-context monitoring implementation. As a result, the following cases will be tested in the order in which they are listed below.

1. System Initialization: This case tests the output of the interrupts when the system initializes (i.e., the reset condition). If implemented correctly, neither interrupt will be triggered.
2. Writing to an Arbitrary Memory Location: This case tests the output of the interrupts when a write request is made to an address other than the address where the PID is stored. If implemented correctly, neither interrupt will be triggered.
3. Reading From an Arbitrary Memory Location: This case tests the output of the interrupts when a write request is made to an address other than the address where the PID is stored. If implemented correctly, neither interrupt will change from their previous state.
4. Reading the PID as a Non-monitored Process Executes: This case tests the output of the interrupts when a memory location is read and a monitored process is not executing (i.e., when both interrupts are “0”). If implemented correctly, neither interrupt will change from their previous state.
5. Writing the First Monitored PID to the PID Address: This case tests the output of the interrupts when a PID corresponding to Monitored Process #1 is written

to the PID address. If implemented correctly, interrupt A will be triggered, while interrupt B will be not be triggered.

6. Writing the Second Monitored PID to the PID Address: This case tests the output of the interrupts when a PID corresponding to Monitored Process #2 is written to the PID address. If implemented correctly, interrupt A will not be triggered, while interrupt B will be triggered.
7. Reading the PID as a Monitored Process Executes: This case tests the output of the interrupts when a memory location is read while a monitored process is executing (i.e., when one interrupt signal is “1”). If implemented correctly, both interrupt signals should not change from their previous state.
8. Writing a Monitored PID to an Arbitrary Memory Location as a the Other Monitored Process Executes: This case tests the output of the interrupts when a value matching the PID of a monitored process is written to a memory location while the other monitored process is executing (i.e., when one interrupt signal is “1”). If implemented correctly, both interrupt signals should not change from their previous state.
9. Writing a Non-monitored PID after a Monitored Process has Been Executing: This case tests the output of the interrupts when a non-monitored PID is written to the PID address directly following the execution of a monitored process. If implemented correctly, neither interrupt will be triggered.

Although this is not an exhaustive test set, the number and type of tests chosen are sufficient to prove the functionality of the multi-context monitoring implementation.

5.2.2 Simulation. Simulating the implementation of the multi-context monitoring concept was performed only for the logic implementing the monitor. The embedded system connected to this logic in our implementation was not part of the simulation, since the functionality of the multi-context monitoring can be tested without the use of the embedded system containing the PPU.

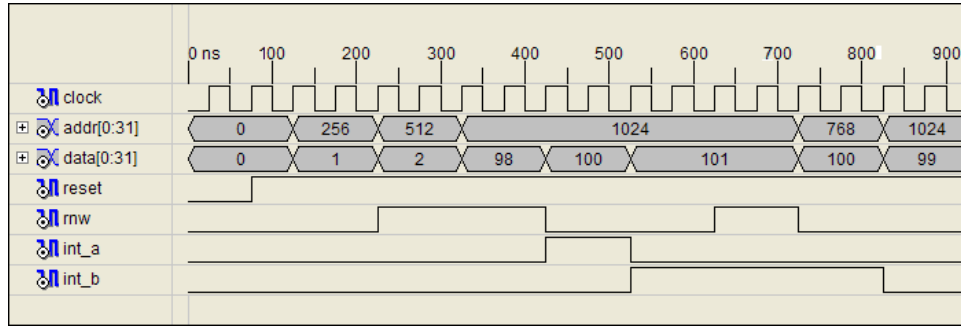


Figure 5.2: Multicontext Monitors Simulation Result

Figure 5.2 displays the simulation waveforms for our multi-context monitoring implementation. Inputs include *clock*, *addr*, *data*, *reset*, and *rnw*. The *addr* and *data* waves represent the address and data lines of the OPB respectively. The *rnw* wave is the write enable line of the OPB, where a “0” represents a write to a particular memory address and a “1” represents a read from a particular memory address. The *clock* and *reset* signals are self explanatory. Outputs include *int_a* and *int_b* which represent the interrupts for Monitored Process #1 and Monitored Process #2, respectively.

Each of the test cases mentioned previously in Section 5.2.1 correlate directly to a 150ns slice of time in the simulation depicted in Figure 5.2. The address where the PID resides is designated as 1024. All other addresses are arbitrary. Data values representing PIDs range from 99-101, however, 100 corresponds to Monitored Process #1 and 101 corresponds to Monitored Process #2. By following the test cases described above, it can be seen that the all test cases are fulfilled. As a result, this simulation has shown that the implementation of our multi-context monitoring concept can determine the currently executing process and keeps track of it in the event of other kinds of memory accesses. It should also be mentioned that the interrupt signals - *int_a* and *int_b* - both update in under one clock cycle when the PID changes.

5.2.3 Implementation Results. Although we succeeded in implementing our multi-context monitoring concept on the development platform, we were not able to perform any real-world test cases. This derived from difficulties with the development tools which did not allow for the design to be implemented given our time constraints. Furthermore, once a change is made to a design, the bitstream used to program the FPGA must be regenerated. Designs including an embedded Microblaze (as this implementation does) required 20-25 minutes to generate the FPGA bitstream. Although subsequent bitstream generations are faster than the original bitstream generation (as some steps in the process do not need to be repeated), the debug cycle is still long at 15 - 20 minutes to modify a bitstream. This somewhat faster bitstream generation could not always be taken advantage of, however, as issues with

the development tools required that the project be “cleaned” often, resulting in all steps of bitstream regeneration having to be redone. This increased the average time required for a single debug cycle.

5.3 Peripheral Access Control

The peripheral access control concept, described in 4.4.3, was not able to be implemented on the development platform. This was not due to any particular difficulty introduced by this monitoring concept, but rather, it can be attributed simply to time constraints. Despite not being able to implement this concept, we still discuss the expected results of this primitive based upon the results of other primitives tested.

5.3.1 Expected Results. The peripheral access control concept uses the same techniques to gather the PID as the multi-context monitoring implementation presented in Section 4.4.2. As it was shown in Section 5.2 that the PID could be captured, there should be no reason as to why such a method would not work for the purpose of implementing peripheral access control. Additionally, the peripheral access table and the peripheral map table proposed for this concept can be implemented in much the same way as the execution policy enforcement module that is described in Section 4.4.1. As the execution policy enforcement module was shown to work in both simulation and implementation in 5.1, it is safe to assume that the peripheral access table and the peripheral map table can be implemented in a similar manner.

Furthermore, despite simulations showing the execution policy enforcement and multi-context monitoring implementations responding in under one clock cycle, we cannot know for certain how fast peripheral access control will respond without implementation and simulation/testing. However, we expect peripheral access control to respond in no more than a few clock cycles as the concept is only mildly more complex than the techniques used to implement execution policy enforcement and multi-context monitoring.

5.4 Asymmetrically Shared Main Memory

The design of the asymmetrically shared main memory concept, detailed in Section 4.4.4, was successfully implemented on our development platform. Time constraints, however, prevented testing from being able to be performed. Since the implementation of this primitive is based on using a Microblaze as the SMU, not RTL as is the case with the other primitives discussed thus far, simulation was not performed. Accordingly, we present our planned testing methodology and the expected results below.

5.4.1 Testing Methodology. Implementation of the asymmetrically shared main memory concept, detailed in Section 4.4.4, a number of memory regions with varying permission levels. Depending on the processor accessing memory at a given time, each region has different permissions that dictate what kind of operations can be performed. As a result, testing is functional in nature and focuses on ensuring that

the permission level of the various regions are correctly enforced for both the PPU and the SMU. We describe the applicable test cases below.

1. PPU Accessing PPU Memory Space: This case tests whether or not the PPU can read data, write data, and execute instructions from its own memory space. If implemented correctly, the PPU will be able to read, write, and execute from its own memory space.
2. PPU Accessing SMU's Memory Space: This case tests whether or not the PPU can access the SMU's memory space using any type of memory access (read/write/execute). If implemented correctly, the PPU will not be able to access the SMU's memory space.
3. SMU Accessing SMU Exclusive Memory Space: This case tests whether or not the SMU can read data, write data, and execute instructions from the SMU Exclusive memory region. If implemented correctly, the SMU will be able to read, write, and execute from this region of memory.
4. SMU Reading/Writing Safe SMU Exclusive Memory Space: This case tests whether or not the SMU can read and write to the Safe SMU Exclusive memory region. If implemented correctly, the SMU will be able to read from and write to this region of memory.
5. SMU Executing From Safe SMU Exclusive Memory Space: This case tests whether or not the SMU can execute instructions from the Safe SMU Exclusive

memory region. If implemented correctly, the SMU will not be able to execute from this region of memory.

6. SMU Reading/Writing PPU Shared Memory Space: This case tests whether or not the SMU can read and write to the PPU Shared Memory region. If implemented correctly, the SMU will be able to read from and write to this region of memory.
7. SMU Executing From PPU Shared Memory Spaces: This case tests whether or not the SMU can execute instructions from the PPU Shared Memory region. If implemented correctly, the SMU will not be able to execute from this region of memory.

5.4.2 Implementation Results. The asymmetrically shared main memory was successfully implemented on the development platform, however, testing was not performed due to time constraints. Despite this, we are confident that our implementation of an asymmetrically shared memory will function correctly. This is because the various regions of our asymmetrically shared memory concept are all enforced in hardware. For example, the PPU connects to the MPMC2 via a single OPB, while the SMU connects to the MPMC2 via two independent and separate OPBs - one for the instruction-side and the other for the data-side. As we have set the PPU's accessible memory range to not include the SMU's memory space, it is not possible for the PPU to even address the SMU's memory space. Thus, by virtue of how memory mapping works and how we have leveraged the Harvard architecture of the Microblaze proces-

sor to implement the regions of varying permissions, we expect with a high level of certainty that the asymmetrically shared main memory implementation will function correctly.

5.5 Memory Management Unit Co-opting

The MMU co-opting concept, detailed in Section 4.4.5, is a large departure from traditional computer architecture design; no processor available, either in the development platform or the computing industry, contains such a capability. As a result, the MMU co-opting concept was not able to be implemented, either in simulation nor in physical hardware. However, we do realize that our MMU concept has a number of benefits as well as limitations. These are described below.

5.5.1 Benefits and Limitations. The MMU co-opting technique can theoretically provide the SMU access to the virtual memory space of a PPU process at the hardware level. As a result, the SMU, not the PPU, controls the MMU co-opting process. This, combined with the design decision to allow the SMU to co-opt the PPU's MMU only when the PPU is either not using the MMU or the SMU has halted the PPU, keeps the SMU invisible from the PPU. Additionally, as this is being implemented at the hardware level, this method can be used to gain insight into state information unable to be gathered at the hardware level previously. This allows for efficient parallel monitoring that can be performed in real-time as code executes on the PPU. As a result, overhead associated with monitoring is expected to be reduced compared to software-based methods. Additionally, as the SMU leverages the physi-

cal addresses it retrieves from the PPU's MMU, this assumes that the SMU not have an MMU of its own. This makes the MMU co-opting concept ideal for situations where chip area is at a premium, while still providing visibility into a process' virtual memory space.

Co-opting the PPU's MMU is not without its limitations, however. One such limitation of our MMU co-opting concept is that the control logic required to implement such a capability may be significant. Additionally, as data in question is accessed by the SMU after the PPU resumes operation, the physical address corresponding to the data structure in question may change. This is a result of the dynamic nature of virtual memory. To remedy this, the physical address of the data structure in question should be retrieved every time the process begins and when the virtual address in question is accessed. This could lead to inefficiencies as the SMU would be requesting translation for a virtual address whose corresponding physical address has not changed. This could be remedied by either retrieving the data when the MMU is co-opted or by making the SMU access the physical memory address before the PPU is allowed to resume operation. However, as the PPU may be disabled while the SMU co-opts the MMU, both of these methods could increase the amount of time that the PPU must remain halted, decreasing the performance of the PPU. Such performance is dependent on the how often the SMU co-opts the MMU and how often the monitored process on the PPU must access memory. Additionally, depending on the data being monitored by the SMU, the data may have been moved to external memory. As the PPU's OS keeps track of where such data would reside, the co-opted MMU

would have no access to such data since the PPU (and the OS) would be halted while the SMU co-opts the PPU.

5.6 Multiple Memory Management Units

As with MMU co-opting, incorporating multiple MMUs into the SMU as presented in Section 4.4.6 is a significant departure from traditional computer architecture design. This combined with time constraints did not allow for either a simulation model or implementation to be created. Even without this, we can discuss a number of the the apparent benefits and limitations of the proposed multiple MMU approach below.

5.6.1 Benefits and Limitations. As with the MMU co-opting concept discussed in Section 5.5, implementing multiple MMUs can provide the SMU visibility into the PPU's virtual memory space. Doing so by integrating multiple MMUs can provide a number of benefits. The most notable of these benefits is the ability to monitor the processes that are not currently executing. This can provide for run-time trustability as this method could be used to ensure a process is in a known trusted state every time before it is placed on the PPU to be executed. Moreover, an SMU with multiple MMUs can also allow for monitoring in different scenarios. Examples of such scenarios include monitoring a process on the I/O queue for buffer overflow attacks or monitoring multiple processes on a waiting queue to check for deadlock conditions.

Monitoring the PPU's virtual memory space with multiple MMUs also shifts the resource burden from the PPU to the SMU. As a result, the PPU does not need to undergo many architectural changes, especially when compared to the changes to the PPU required to implement MMU co-opting. Moreover, as the SMU monitors the PPU's virtual memory space using its own hardware resources, the performance of the PPU should not be affected, as would most likely be the case with MMU co-opting (due to having to halt the processor's execution when co-opting the MMU). As a result, impacts on system usability should be minimal.

Despite all of the benefits that an SMU with multiple MMUs can provide, a number of limitations still exist. While having the SMU utilize its own resources may allow for better PPU performance and fewer modifications to the PPU's architecture, the SMU has to undergo a drastic architectural change that may require a large amount of added complexity to the SMU's architecture. This may not turn out to be the case, however, as the SMU may not need to be very powerful for the kinds of monitoring it will be performing, but it is still a notable limitation nonetheless.

A number of the capabilities provided by the SMU having multiple MMUs are also dependent on a level of PPU software support. State information, such as the page directory address of different processes or information regarding what processes reside on which OS queues, must be explicitly communicated from the PPU to the SMU. This explicit communication should be kept to a minimum. This is because explicit communication not only means that the PPU must be aware of the SMU's presence, but that some of the SMU's capabilities may be dependent on the PPU's

potentially untrustable software. If not handled in a careful manner, such as the PPU only explicitly communicating to the SMU when the PPU's OS is in a known trusted state, then it will be easier for the SMU to be compromised.

Another limitation that must be considered is that the OS controls the paging of data that no longer resides in memory. As part of paging is controlled by the operating system for pages residing in external storage (no longer in main memory), the OS may have protections to keep data belonging to a currently non-executing process from being accessed.

VI. Conclusion

In this Chapter, we present the key findings of our research. Additionally, we describe the research areas we would like to explore in the future.

6.1 Conclusions

Our research focuses on moving security-related monitoring tasks from software to dedicated hardware in an effort to increase overall system security and usability compared to software-based security methods. This is realized via a number of hardware-based functional primitives that gather and process state information in ways not previously possible at the hardware-level. These primitives leverage a novel computing architecture that is based on a contemporary shared memory multiprocessing model. In doing this, we are able to break through a number of limitations imposed by the current computing model, resulting in framework upon which real-time security policy compliance monitoring can be performed in parallel and for a wide variety of computing environments. As we show that performing security policy compliance monitoring in this manner can increase performance, efficiency, and security over software-based methods, we validate our research hypothesis. The key findings that allow us to make this claim are presented below.

6.1.1 Improved Time-to-Detect. Our research has shifted security-related monitoring tasks from software to dedicated hardware. Thus, security monitoring can be performed in parallel as code executes on the monitored processor. This, combined with gathering state information at the hardware level, provides for real-time security

policy compliance monitoring. As a result, our research allows for faster time-to-detect than software-based methods. Thus, damage caused by malicious events can be minimized or prevented altogether.

6.1.2 Hardness of the Monitoring System. Software is the primary means of attacking a system. Therefore, we designed the architecture to tightly couple the monitoring hardware to the monitored system, while minimizing software coupling as much as possible. This allows the monitoring hardware to gather context-rich state information, but do so with a minimal amount of explicit communication from the monitored system. There are two key benefits that come from this: 1) the attack surface of the monitoring hardware is decreased, making the monitoring hardware more secure, and 2) The monitoring hardware can continue to function, albeit possibly in a diminished capacity, in the event that the monitored system is compromised. Thus, the monitoring system itself is highly resistant to being compromised.

6.1.3 Displaced Security Workload. Our research shifts the burden of performing security monitoring tasks to dedicated hardware. Thus, security monitoring can be performed in parallel as code executes on the monitored processor. As a result, little to no overhead (due to security monitoring) is incurred by the processor executing the monitored code. This increases the performance of the monitored system compared to software-based approaches. Therefore, little to no impact on the system's usability occurs.

6.1.4 Novel Hardware-based Monitoring Techniques. Our architecture allows state information previously not available at the hardware level to be gathered. This increases the types of inputs available to the monitoring system over previously proposed hardware-based security monitoring systems. Moreover, this new state information enables novel monitoring capabilities at the hardware level. We describe the benefits of such capabilities below.

- **Multi-context Monitoring in Hardware:** By monitoring the the PID of the currently executing process, monitors implemented in hardware can now discern between different processes. This allows hardware-based monitors to be able to operate in dynamic, multiprogrammed (e.g., general purpose) environments, rather than be limited to more static (e.g., embedded and application specific) environments. As a result, we provide an alternative to anomaly detection (which is prone to false positives) when performing hardware-based monitoring in more complex computing environments.
- **Virtual Memory Introspection:** The MMU co-opting and the Multiple MMU primitives can be used to monitor the virtual memory space of the process currently executing on the monitored processor. This allows both kernel-level and user-level processes to be monitored. Such a capability was previously only available via software-based monitoring techniques, which introduced overhead and was vulnerable to attack. By monitoring the virtual memory space via

hardware, the performance, time-to-detect, and monitoring system security is improved compared to similar software-based methods.

- **Monitoring Non-executing Processes:** Implementing monitoring hardware using multiple MMUs enables the monitoring of any process' virtual memory space; including the memory space of processes not currently executing (i.e., processes residing in one of the OS's waiting queues). This provides completely new security monitoring capabilities. Such capabilities include, but are not limited to, ensuring a process is trustable throughout its entire run-time, bad I/O detection, and run-time deadlock detection - all of which benefit the security-related monitoring field.

6.1.5 Monitoring System Flexibility. Many of the primitives created through the course of our research provide complementary monitoring capabilities. Thus, as the monitoring system (i.e., the SMU) is seen as a black box with respect to the rest of the system, the monitoring system can consist of a combination of primitives. As a result, the monitoring system is flexible and allows security to be tailored to a particular system's specific security needs.

6.1.6 Monitoring System Extensibility. Since the monitoring system (i.e., the SMU) can be viewed as a black box with respect to the rest of the system, any primitive can be implemented, providing it adheres the guidelines of the architecture. As a result, the primitives that can be implemented in this architecture are not limited

to the primitives developed in this research. Thus, new primitives can be developed in the future to enable new forms of security related monitoring.

6.1.7 Improved Range of Monitoring Granularity. Our primitives can allow monitoring granularity ranging from the individual instruction level to the process level. Thus, this research increases the range of monitoring granularity that can be provided via hardware-based mechanisms. This allows the primitives to provide security policy compliance monitoring in a broad range of computing environments, rather than being limited to a single computing environment.

6.2 Future Work

We have determined a number of capabilities that can be provided by the concepts proposed in this research, but that are outside the scope of our primary research goals. As a result, there are a number of areas we would like to explore in future research efforts. These areas are described below.

6.2.1 Virtual Memory Introspection Implementation. The platform that was used for prototyping our systems can implement both PowerPC 405 processors and Xilinx Microblaze processors. The PowerPC cores contain an MMU, however they are hardcores and cannot be modified. The Microblaze cores, are softcores, hence they are modifiable, however they do not contain an MMU of their own. As a result, we were not able to implement our co-opted MMU or SMU with multiple MMUs concepts presented in Sections 3.5.4.2 and 3.5.4.3, respectively. Implementing

such capabilities would greatly benefit from having access to a softcore processor that contains an MMU. This leaves two options: 1) Gain access to the source code for the Microblaze processor and modify it to include an MMU, or 2) use a softcore processor that contains an MMU.

As the Microblaze is based on a Harvard architecture, Option 1 above may be more difficult as there are separate data and instruction buses. Option 2, however, may be able to be realized with the Leon3 softcore processor. As mentioned in Section 4.2.1.3, the Leon 3 is a softcore processor with an MMU. As a result, it may be possible to use the Leon3 processor to prototype our MMU co-opting and multiple MMU concepts.

6.2.2 Enhanced Debug Registers. Contemporary processors include registers that are used to monitor a number of memory addresses for debugging purposes. Debug registers can typically monitor both addresses and data and can be set to trigger on varying conditions. As such, this kind of capability may be useful for security related monitoring tasks. For example, a breakpoint could be set for a particular address containing a key invariant. The debug registers could then be used to trigger a signal when the memory address is accessed. Rather than halting the processor, the triggered breakpoint could be used to signal the SMU to notify it of the event and have it perform an invariant check. Adding this enhanced debug capability would most likely not be very difficult as it would consist mostly of tapping into the memory bus, which we have already shown to work in Section 4.4.2.

Additionally, many processor architectures only contain a small number of debug registers (Intel processors have four [26]), however, if using such a capability for security-related monitoring rather than debugging, it may be desired to have more of these registers. Instead, 25 or so could be implemented to allow for monitoring a large number of events that would indicate malicious activity. By doing this, the SMU would respond to specific events as they happen, rather than periodically checking a number of locations to see if an invariant has changed. Moreover, these debug registers could be enhanced to monitor a range of addresses, rather than just a single address. This would be beneficial as one could monitor over a broad range of addresses, making it more likely that the event will be detected. This could possibly be helpful for either detecting previously unencountered malicious activity or for malicious activity that does not always work on the same memory address.

6.2.3 Forensics Capabilities. The various memory introspection methods provide for visibility into both the physical and virtual memory spaces. While the primary goal of such a capability is real-time monitoring of key invariants in order to detect illegitimate activity, it also could provide a convenient platform for data forensics capabilities. As such, key portions of monitored code can not only be monitored, but stored as well. This information could then be used at a later time to analyze attacks and find security holes within the monitored process. This can be extended to include state information gathered from other system resources other than memory such as the PC or the process' PID. Correlating this state information with

data gathered from a monitored process' memory may be able to provide for forensics capabilities not currently available.

6.2.4 Automatic Process Repair. While our memory introspection techniques focus on reading the state of a process' memory space (physical and virtual) in hardware, these memory introspection primitives provide the ability to write to a process' memory space as well. This could potentially be used for repairing processes in the event that they are damaged from an attack. As we are monitoring state information in realtime, it may even be possible to detect and repair such damage automatically, providing a powerful self-healing capability in real-time. An SMU containing multiple MMUs, as was presented in Sections 3.5.4.3 and 4.4.6, would be particularly suited to such a capability since process damage could be repaired and detected while a process resides on the ready, I/O, or waiting queues. However, writing to a monitored process' memory space can be dangerous for the monitored process, as a completely separate process (executing on the SMU) can potentially damage the monitored process if done incorrectly. As a result, care must be taken so as to not further damage the monitored process.

6.2.5 Minimum Required Resource Investigation. The implementation of our primitives was done as a proof of concept. As such, we did not look at resource usage in terms of area, power, etc. Thus, the primitives are probably using more resources than are required for the tasks they perform. For example, the SMU may not need a powerful processor to perform memory introspection. Thus it would be

interesting to investigate the minimum resource requirements for each capability to improve efficiency. This would help when integrating our concepts into commercial designs as chip-area is at a premium. As there are current efforts to move dedicated coprocessing tasks to the CPU packaging and eventually onto the core itself with AMD’s Torrenza initiative [44], our security primitives could be one such form of coprocessing. Moreover, a small portion of a processor could include FPGA fabric to allow our primitives to be configured directly on the CPU and tailored to a specific application. Such an application would greatly benefit from determining how such concepts can be implemented while using resources efficiently.

6.2.6 Scalability: Multiple PPU’s per SMU. The architecture we propose currently allows for one SMU monitoring each PPU in the system. However, the PPU may not always be executing a monitored process, leaving the SMU unused. As a result, the SMU may be able to use its resources to monitor other PPU’s within the system that may be executing a monitored process. Having a single SMU service multiple PPU’s would be a more efficient use of resources and serve to minimize the chip area devoted to SMUs, allowing for more chip resources to be devoted to the PPU’s. While the SMU could potentially be switched to only work with one PPU at a time, it may be possible for a single SMU to service multiple PPU’s simultaneously, so long as the required hardware primitives are orthogonal to each other. For example, the PC monitor could be monitoring one PPU, while the memory introspection tasks could be performed on another PPU. The focus of such research would be the inter-

connection method between the PPU and the SMU, as well as the software model of the SMU itself.

6.2.7 Security Logic Units. Microprocessors are already moving toward multicore designs and there seem to be no end in sight with Sun Microsystem's Ultra-SPARC T1 processor having 8 cores now [54]. Additionally, recent research conducted by Intel has produced what they are terming as "Tera-scale processors" [21]. These processors further the current multicore paradigm by integrating a large number (80 cores for the tera-scale prototype) of simple cores in order to increase performance. All of these cores would need some form of security monitoring, therefore, it may be possible to abstract away the concept of an SMU, and make each primitive into a type of security logical unit (SLU) that focuses on a particular type of data processing. The entire processor would have access to a number of each kind of SLU, similar to how processor pipelines today have access to a number of multipliers, adders, etc. Depending on the code executing on the particular core, it could use whatever SLUs are needed at the particular time for SPCM purposes. Reconfigurable logic could even be used to provide for varying types and numbers of SLUs depending on the current application being monitored. This organization of security resources would make it difficult for the monitoring primitives to remain transparent to the processing cores being monitored, thus new hardware-based mechanisms would most likely need to be proposed that could ensure the security and proper operation of the SLUs.

Appendix A. Implementation Code

A.1 Execution Policy Enforcement Module

A.1.1 MB_Trace4mdm_top.vhd.

```
-----
-- Created by: 2Lt Stephen Mott, USAF
-- Create Date: 13:23:25 02/06/2007
-- Design Name: Execution Policy Enforcement Module
-- Module Name: MB_Trace4mdm_top
-- Project Name: mbtrace4projmdm.ise
-- Target Devices: XilinxML310 development board
-- Tool versions: 8.2i
--
-- Description: This is the top-level structural definition of our
--              execution policy enforcement system. It connects
--              the non-executable memory table (noex_mem) and the
--              non-executable memory enable logic (noex_mem_en)
--              to the embedded system containing the PPU (i.e. a
--              microblaze processor) that was created in EDK.
--
-- Dependencies: This module requires the use of the
--               system.xmp from the mb_trace4_int_mdm_test EDK
--               project. Also required are the noex_mem_en.vhd
--               and noex_mem.xco modules.
-----

library IEEE; use IEEE.STD_LOGIC_1164.ALL; use
IEEE.STD_LOGIC_ARITH.ALL; use IEEE.STD_LOGIC_UNSIGNED.ALL; entity
MB_Trace4mdm_top is --these are pins that connect to signals on the
ML310 board itself PORT(
    fpga_0_RS232_Uart_RX_pin : IN std_logic;
    fpga_0_SysACE_CompactFlash_SysACE_CLK_pin : IN std_logic;
    fpga_0_SysACE_CompactFlash_SysACE_MPIRQ_pin : IN std_logic;
    sys_clk_pin : IN std_logic;
    sys_rst_pin : IN std_logic;
    fpga_0_LEDs_8Bit_GPIO_IO_pin : INOUT std_logic_vector(0 to 7);
    fpga_0_LCD_OPTIONAL_GPIO_IO_pin : INOUT std_logic_vector(0 to 11);
    fpga_0_SysACE_CompactFlash_SysACE_MPD_pin : INOUT std_logic_vector(7 downto 0);
    fpga_0_RS232_Uart_TX_pin : OUT std_logic;
    fpga_0_SysACE_CompactFlash_clk_enable_n_pin : OUT std_logic;
```

```

        fpga_0_SysACE_CompactFlash_SysACE_MPA_pin : OUT std_logic_vector(6 downto 0);
        fpga_0_SysACE_CompactFlash_SysACE_CEN_pin : OUT std_logic;
        fpga_0_SysACE_CompactFlash_SysACE_OEN_pin : OUT std_logic;
        fpga_0_SysACE_CompactFlash_SysACE_WEN_pin : OUT std_logic;
        fpga_0_ORGate_1_Res_pin : OUT std_logic;
        fpga_0_ORGate_1_Res_1_pin : OUT std_logic;
        fpga_0_ORGate_1_Res_2_pin : OUT std_logic);
end MB_Trace4mdm_top;

--component definitions
architecture Structural of MB_Trace4mdm_top is

--this defines the inputs and outputs of the microblaze system
created in the EDK COMPONENT system PORT(
    fpga_0_RS232_Uart_RX_pin : IN std_logic;
    fpga_0_SysACE_CompactFlash_SysACE_CLK_pin : IN std_logic;
    fpga_0_SysACE_CompactFlash_SysACE_MPIRQ_pin : IN std_logic;
    sys_clk_pin : IN std_logic;
    sys_rst_pin : IN std_logic;
    microblaze_0_INTERRUPT_pin : IN std_logic;
    sys_clk_s_pin : IN std_logic;
    fpga_0_LEDs_8Bit_GPIO_IO_pin : INOUT std_logic_vector(0 to 7);
    fpga_0_LCD_OPTIONAL_GPIO_IO_pin : INOUT std_logic_vector(0 to 11);
    fpga_0_SysACE_CompactFlash_SysACE_MPD_pin : INOUT std_logic_vector(7 downto 0);
    fpga_0_RS232_Uart_TX_pin : OUT std_logic;
    fpga_0_SysACE_CompactFlash_clk_enable_n_pin : OUT std_logic;
    fpga_0_SysACE_CompactFlash_SysACE_MPA_pin : OUT std_logic_vector(6 downto 0);
    fpga_0_SysACE_CompactFlash_SysACE_CEN_pin : OUT std_logic;
    fpga_0_SysACE_CompactFlash_SysACE_OEN_pin : OUT std_logic;
    fpga_0_SysACE_CompactFlash_SysACE_WEN_pin : OUT std_logic;
    fpga_0_ORGate_1_Res_pin : OUT std_logic;
    fpga_0_ORGate_1_Res_1_pin : OUT std_logic;
    fpga_0_ORGate_1_Res_2_pin : OUT std_logic;
    system_clk_pin : OUT std_logic;
    microblaze_0_Trace_PC_pin : OUT std_logic_vector(0 to 31));
END COMPONENT;

--these are the pins of the no-ex_mem module
component noex_mem
    port (
        clka: IN std_logic;
        addra: IN std_logic_VECTOR(14 downto 0);
        ena: IN std_logic;

```



```

        douta: OUT std_logic_VECTOR(0 downto 0));
end component;

-- FPGA Express Black Box declaration for creating the noex_mem
module attribute fpga_dont_touch: string; attribute fpga_dont_touch
of noex_mem: component is "true";

-- Synplicity black box declaration for creating the noex_mem module
attribute syn_black_box : boolean; attribute syn_black_box of
noex_mem: component is true;

--these are the pins of the no-ex_mem_en module COMPONENT
noex_mem_en PORT(
    CLK : IN std_logic;
    PC_in : IN std_logic_vector(31 downto 15);
    Enable : OUT std_logic);
END COMPONENT;

--signals for connecting instantiated components signal CLOCK :
STD_LOGIC; signal MB_INT : STD_LOGIC_VECTOR(0 DOWNT0 0); signal
PC_BITS : STD_LOGIC_VECTOR(31 DOWNT0 0); signal ENABLE_MEM :
STD_LOGIC;

begin --the different components are instantiated and connected below

--instantiation and port mapping of the
--embedded system created using the EDK
system_i: system PORT MAP(

fpga_0_RS232_Uart_RX_pin => fpga_0_RS232_Uart_RX_pin,
fpga_0_RS232_Uart_TX_pin => fpga_0_RS232_Uart_TX_pin,
fpga_0_LEDs_8Bit_GPIO_IO_pin => fpga_0_LEDs_8Bit_GPIO_IO_pin,
fpga_0_LCD_OPTIONAL_GPIO_IO_pin => fpga_0_LCD_OPTIONAL_GPIO_IO_pin,
fpga_0_SysACE_CompactFlash_SysACE_CLK_pin =>
    fpga_0_SysACE_CompactFlash_SysACE_CLK_pin,
fpga_0_SysACE_CompactFlash_clk_enable_n_pin =>
    fpga_0_SysACE_CompactFlash_clk_enable_n_pin,
fpga_0_SysACE_CompactFlash_SysACE_MPA_pin =>
    fpga_0_SysACE_CompactFlash_SysACE_MPA_pin,
fpga_0_SysACE_CompactFlash_SysACE_MPD_pin =>
    fpga_0_SysACE_CompactFlash_SysACE_MPD_pin,
fpga_0_SysACE_CompactFlash_SysACE_CEN_pin =>

```

```

    fpga_0_SysACE_CompactFlash_SysACE_CEN_pin,
fpga_0_SysACE_CompactFlash_SysACE_OEN_pin =>
    fpga_0_SysACE_CompactFlash_SysACE_OEN_pin,
fpga_0_SysACE_CompactFlash_SysACE_WEN_pin =>
    fpga_0_SysACE_CompactFlash_SysACE_WEN_pin,
fpga_0_SysACE_CompactFlash_SysACE_MPIRQ_pin =>
    fpga_0_SysACE_CompactFlash_SysACE_MPIRQ_pin,
fpga_0_ORGate_1_Res_pin => fpga_0_ORGate_1_Res_pin,
fpga_0_ORGate_1_Res_1_pin => fpga_0_ORGate_1_Res_1_pin,
fpga_0_ORGate_1_Res_2_pin => fpga_0_ORGate_1_Res_2_pin,
sys_clk_pin => sys_clk_pin,
sys_rst_pin => sys_rst_pin,
microblaze_0_INTERRUPT_pin => MB_INT(0),
system_clk_pin => CLOCK,
sys_clk_s_pin => CLOCK,
microblaze_0_Trace_PC_pin => PC_BITS);

--instantiation and port mapping of the --noex_mem module
noex_mem_module : noex_mem
    port map (
        clka => CLOCK,
        addra => PC_BITS(14 DOWNT0 0),
        ena => ENABLE_MEM,
        douta => MB_INT);

--instantiation and port mapping of the --noex_mem_en module
noex_mem_en_module : noex_mem_en PORT MAP(
    CLK => CLOCK,
    PC_in => PC_BITS(31 DOWNT0 15),
    Enable => ENABLE_MEM);
end Structural;

```

A.1.2 noex_mem_en.vhd.

```

-- Created By:      2Lt Stephen Mott, USAF
-- Create Date:     13:37:46 02/06/2007
-- Design Name:     Execution Policy Enforcement Module
-- Module Name:     noex_mem_en_module - Behavioral
-- Project Name:    mbtrace4projmdm.ise
-- Target Devices:  XilinxML310 development board
-- Tool versions:   8.2i
--

```

```

-- Description: This module is a behavioral definition of the enable
--               logic for the noex_mem module.  As it may be
--               desired that only certain regions of memory may
--               want to be monitored, we created this enable logic
--               to allow monitoring to only occur for particular
--               memory addresses.  Thus, any PC within the range,
--               will activate the monitor(i.e. the noex_mem BRAM).
--               Our enable logic currently provides for program
--               counter values below 0x00008000 to enable the
--               noex_mem module.
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity noex_mem_en is
    Port (
        CLK : in  STD_LOGIC;           --clock signal input
        PC_in : in  STD_LOGIC_VECTOR (31 downto 15); --upper 17 bits of
                                                --the microblaze PC
        Enable : out  STD_LOGIC);       --enable signal output
end noex_mem_en;

architecture Behavioral of noex_mem_en is
begin
    process(CLK)
    begin
        if(PC_in = "000000000000000000") then --sets enable signal if PC is below
            Enable <= '1';                     --address 0x00008000.
        else
            Enable <= '0';
        end if;
    end process;
end Behavioral;

```

A.2 Multi-context Hardware Monitoring

A.2.1 MB_PID2_top.vhd.

```

-----
-- Created By:      2Lt Stephen Mott, USAF
-- Create Date:     09:01:53 01/24/2007
-- Design Name:     Multi-context hardware monitoring
-- Module Name:     MB_PID2_top - MB_PID2_Struct
-- Project Name:    MB_PID2.isc
-- Target Devices:  Xilinx ML310 development board
-- Tool versions:   8.2i

-- Description:     This is the top-level structural definition of our Multi-context
--                  monitors system. It connects the PID retrieval logic to the
--                  embedded system that contains the PPU (i.e. a microblaze
--                  processor).
--
-- Dependencies:    requires the system.xmp file in the MB_PID2 EDK project and the
--                  PID_Logic.vhd file.
--
-----

```

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

```

```

entity MB_PID2_top is

```

```

--these are pins that connect to signals on the ML310 board itself

```

```

PORT(
    fpga_0_RS232_Uart_RX_pin : IN std_logic;
    fpga_0_SysACE_CompactFlash_SysACE_CLK_pin : IN std_logic;
    fpga_0_SysACE_CompactFlash_SysACE_MPIRQ_pin : IN std_logic;
    fpga_0_DDR_CLK_FB : IN std_logic;
    sys_clk_pin : IN std_logic;
    sys_rst_pin : IN std_logic;
    fpga_0_DDR_SDRAM_32Mx64_DDR_DQS_pin :
        INOUT std_logic_vector(0 to 3);
    fpga_0_DDR_SDRAM_32Mx64_DDR_DQ_pin :
        INOUT std_logic_vector(0 to 31);
    fpga_0_LEDs_8Bit_GPIO_IO_pin :
        INOUT std_logic_vector(0 to 7);
    fpga_0_SysACE_CompactFlash_SysACE_MPD_pin :
        INOUT std_logic_vector(7 downto 0);
    fpga_0_DDR_SDRAM_32Mx64_DDR_Clk_pin : OUT std_logic;
    fpga_0_DDR_SDRAM_32Mx64_DDR_Clk_n_pin : OUT std_logic;
    fpga_0_DDR_SDRAM_32Mx64_DDR_Addr_pin :

```

```

        OUT std_logic_vector(0 to 12);
fpga_0_DDR_SDRAM_32Mx64_DDR_BankAddr_pin :
    OUT std_logic_vector(0 to 1);
fpga_0_DDR_SDRAM_32Mx64_DDR_CASn_pin : OUT std_logic;
fpga_0_DDR_SDRAM_32Mx64_DDR_CKE_pin : OUT std_logic;
fpga_0_DDR_SDRAM_32Mx64_DDR_CSn_pin : OUT std_logic;
fpga_0_DDR_SDRAM_32Mx64_DDR_RASn_pin : OUT std_logic;
fpga_0_DDR_SDRAM_32Mx64_DDR_WEn_pin : OUT std_logic;
fpga_0_DDR_SDRAM_32Mx64_DDR_DM_pin : OUT std_logic_vector(0 to 3);
fpga_0_RS232_Uart_TX_pin : OUT std_logic;
fpga_0_SysACE_CompactFlash_clk_enable_n_pin : OUT std_logic;
fpga_0_SysACE_CompactFlash_SysACE_MPA_pin :
    OUT std_logic_vector(6 downto 0);
fpga_0_SysACE_CompactFlash_SysACE_CEN_pin : OUT std_logic;
fpga_0_SysACE_CompactFlash_SysACE_OEN_pin : OUT std_logic;
fpga_0_SysACE_CompactFlash_SysACE_WEN_pin : OUT std_logic;
fpga_0_ORGate_1_Res_pin : OUT std_logic;
fpga_0_ORGate_1_Res_1_pin : OUT std_logic;
fpga_0_ORGate_1_Res_2_pin : OUT std_logic;
fpga_0_DDR_CLK_FB_OUT : OUT std_logic);
end MB_PID2_top;

--component definitions
architecture MB_PID2_Struct of MB_PID2_top is

--this defines the inputs and outputs of the microblaze system created in the EDK
COMPONENT system
PORT(
    fpga_0_RS232_Uart_RX_pin : IN std_logic;
    fpga_0_SysACE_CompactFlash_SysACE_CLK_pin : IN std_logic;
    fpga_0_SysACE_CompactFlash_SysACE_MPIRQ_pin : IN std_logic;
    fpga_0_DDR_CLK_FB : IN std_logic;
    sys_clk_pin : IN std_logic;
    sys_rst_pin : IN std_logic;
    fpga_0_DDR_SDRAM_32Mx64_DDR_DQS_pin :
        INOUT std_logic_vector(0 to 3);
    fpga_0_DDR_SDRAM_32Mx64_DDR_DQ_pin :
        INOUT std_logic_vector(0 to 31);
    fpga_0_LEDs_8Bit_GPIO_IO_pin : INOUT std_logic_vector(0 to 7);
    fpga_0_SysACE_CompactFlash_SysACE_MPD_pin :
        INOUT std_logic_vector(7 downto 0);
    fpga_0_DDR_SDRAM_32Mx64_DDR_Clk_pin : OUT std_logic;
    fpga_0_DDR_SDRAM_32Mx64_DDR_Clk_n_pin : OUT std_logic;

```

```

fpga_0_DDR_SDRAM_32Mx64_DDR_Addr_pin :
    OUT std_logic_vector(0 to 12);
fpga_0_DDR_SDRAM_32Mx64_DDR_BankAddr_pin :
    OUT std_logic_vector(0 to 1);
fpga_0_DDR_SDRAM_32Mx64_DDR_CASn_pin : OUT std_logic;
fpga_0_DDR_SDRAM_32Mx64_DDR_CKE_pin : OUT std_logic;
fpga_0_DDR_SDRAM_32Mx64_DDR_CSn_pin : OUT std_logic;
fpga_0_DDR_SDRAM_32Mx64_DDR_RASn_pin : OUT std_logic;
fpga_0_DDR_SDRAM_32Mx64_DDR_WEn_pin : OUT std_logic;
fpga_0_DDR_SDRAM_32Mx64_DDR_DM_pin : OUT std_logic_vector(0 to 3);
fpga_0_RS232_Uart_TX_pin : OUT std_logic;
fpga_0_SysACE_CompactFlash_clk_enable_n_pin : OUT std_logic;
fpga_0_SysACE_CompactFlash_SysACE_MPA_pin :
    OUT std_logic_vector(6 downto 0);
fpga_0_SysACE_CompactFlash_SysACE_CEN_pin : OUT std_logic;
fpga_0_SysACE_CompactFlash_SysACE_OEN_pin : OUT std_logic;
fpga_0_SysACE_CompactFlash_SysACE_WEN_pin : OUT std_logic;
fpga_0_ORGate_1_Res_pin : OUT std_logic;
fpga_0_ORGate_1_Res_1_pin : OUT std_logic;
fpga_0_ORGate_1_Res_2_pin : OUT std_logic;
fpga_0_DDR_CLK_FB_OUT : OUT std_logic;

--the below pins are the pins used for our external logic to
--tap into the embedded system created in the EDK
Int_ProcessA_pin : IN std_logic;           --Interrupt for PID A
Int_ProcessB_pin : IN std_logic;           --Interrupt for PID B
CLK_OUT_pin : OUT std_logic;               --system Clock
RST_OUT_pin : OUT std_logic;               --system reset
OPB_RNW_pin : OUT std_logic;               --OPB read/write signal
OPB_ABus_pin : OUT std_logic_vector(0 to 31); --OPB address lines
OPB_DBus_pin : OUT std_logic_vector(0 to 31); --OPB data lines
Trace_PC_pin : OUT std_logic_vector(0 to 31)); --microblaze program counter
END COMPONENT;

--this defines the inputs and outputs of the logic used to retrieve the PID
COMPONENT PID_LOGIC
    PORT(
        ADDR_IN : IN std_logic_vector(0 to 31);
        DATA_IN : IN std_logic_vector(0 to 31);
        RNW_IN : IN std_logic;
        CLK_IN : IN std_logic;
        RST_IN : IN std_logic;
        INT_A_OUT : OUT std_logic;

```

```

        INT_B_OUT : OUT std_logic);
END COMPONENT;

--connection signals for instantiated components
SIGNAL CLOCK : std_logic;
SIGNAL INT_A : std_logic;
SIGNAL INT_B : std_logic;
SIGNAL ADDR  : std_logic_vector(0 to 31);
SIGNAL DATA : std_logic_vector(0 to 31);
SIGNAL RESET : std_logic;
SIGNAL RNW   : std_logic;

begin --the different components are instantiated and connected below

--instantiation and port mapping of the
--embedded system created using the EDK
system_i : system
PORT MAP(
    fpga_0_DDR_SDRAM_32Mx64_DDR_Clk_pin =>
        fpga_0_DDR_SDRAM_32Mx64_DDR_Clk_pin,
    fpga_0_DDR_SDRAM_32Mx64_DDR_Clk_n_pin =>
        fpga_0_DDR_SDRAM_32Mx64_DDR_Clk_n_pin,
    fpga_0_DDR_SDRAM_32Mx64_DDR_Addr_pin =>
        fpga_0_DDR_SDRAM_32Mx64_DDR_Addr_pin,
    fpga_0_DDR_SDRAM_32Mx64_DDR_BankAddr_pin =>
        fpga_0_DDR_SDRAM_32Mx64_DDR_BankAddr_pin,
    fpga_0_DDR_SDRAM_32Mx64_DDR_CASn_pin =>
        fpga_0_DDR_SDRAM_32Mx64_DDR_CASn_pin,
    fpga_0_DDR_SDRAM_32Mx64_DDR_CKE_pin =>
        fpga_0_DDR_SDRAM_32Mx64_DDR_CKE_pin,
    fpga_0_DDR_SDRAM_32Mx64_DDR_CSn_pin =>
        fpga_0_DDR_SDRAM_32Mx64_DDR_CSn_pin,
    fpga_0_DDR_SDRAM_32Mx64_DDR_RASn_pin =>
        fpga_0_DDR_SDRAM_32Mx64_DDR_RASn_pin,
    fpga_0_DDR_SDRAM_32Mx64_DDR_WEn_pin =>
        fpga_0_DDR_SDRAM_32Mx64_DDR_WEn_pin,
    fpga_0_DDR_SDRAM_32Mx64_DDR_DM_pin =>
        fpga_0_DDR_SDRAM_32Mx64_DDR_DM_pin,
    fpga_0_DDR_SDRAM_32Mx64_DDR_DQS_pin =>
        fpga_0_DDR_SDRAM_32Mx64_DDR_DQS_pin,
    fpga_0_DDR_SDRAM_32Mx64_DDR_DQ_pin =>
        fpga_0_DDR_SDRAM_32Mx64_DDR_DQ_pin,
    fpga_0_RS232_Uart_RX_pin => fpga_0_RS232_Uart_RX_pin,

```

```

fpga_0_RS232_Uart_TX_pin => fpga_0_RS232_Uart_TX_pin,
fpga_0_LEDs_8Bit_GPIO_IO_pin => fpga_0_LEDs_8Bit_GPIO_IO_pin,
fpga_0_SysACE_CompactFlash_SysACE_CLK_pin =>
    fpga_0_SysACE_CompactFlash_SysACE_CLK_pin,
fpga_0_SysACE_CompactFlash_clk_enable_n_pin =>
    fpga_0_SysACE_CompactFlash_clk_enable_n_pin,
fpga_0_SysACE_CompactFlash_SysACE_MPA_pin =>
    fpga_0_SysACE_CompactFlash_SysACE_MPA_pin,
fpga_0_SysACE_CompactFlash_SysACE_MPD_pin =>
    fpga_0_SysACE_CompactFlash_SysACE_MPD_pin,
fpga_0_SysACE_CompactFlash_SysACE_CEN_pin =>
    fpga_0_SysACE_CompactFlash_SysACE_CEN_pin,
fpga_0_SysACE_CompactFlash_SysACE_OEN_pin =>
    fpga_0_SysACE_CompactFlash_SysACE_OEN_pin,
fpga_0_SysACE_CompactFlash_SysACE_WEN_pin =>
    fpga_0_SysACE_CompactFlash_SysACE_WEN_pin,
fpga_0_SysACE_CompactFlash_SysACE_MPIRQ_pin =>
    fpga_0_SysACE_CompactFlash_SysACE_MPIRQ_pin,
fpga_0_ORGate_1_Res_pin => fpga_0_ORGate_1_Res_pin,
fpga_0_ORGate_1_Res_1_pin => fpga_0_ORGate_1_Res_1_pin,
fpga_0_ORGate_1_Res_2_pin => fpga_0_ORGate_1_Res_2_pin,
fpga_0_DDR_CLK_FB => fpga_0_DDR_CLK_FB,
fpga_0_DDR_CLK_FB_OUT => fpga_0_DDR_CLK_FB_OUT,
sys_clk_pin => sys_clk_pin,
sys_rst_pin => sys_rst_pin,
Int_ProcessA_pin => INT_A,
Int_ProcessB_pin => INT_B,
CLK_OUT_pin => CLOCK,
RST_OUT_pin => RESET,
OPB_RNW_pin => RNW,
OPB_ABus_pin => ADDR,
OPB_DBus_pin => DATA,
Trace_PC_pin => open); --"Trace_PC_pin" is included to provide for
                        --the connection of monitoring logic that may be
                        --added to this project in the future. Currently,
                        --it is left unconnected.

```

```

--instantiation and port mapping of the
--PID retrieval logic
PID_LOGIC_inst : PID_LOGIC
    PORT MAP(
        ADDR_IN => ADDR,
        DATA_IN => DATA,

```



```

        RNW_IN  => RNW,
        CLK_IN => CLOCK,
        RST_IN => RESET,
        INT_A_OUT => INT_A,
        INT_B_OUT => INT_B);
end MB_PID2_Struct;

```

A.2.2 *PID_LOGIC.vhd.*

```

-----
-- Created by:      2Lt Stephen Mott, USAF
-- Create Date:     14:10:33 01/24/2007
-- Design Name:     Multi-context Hardware Monitoring
-- Module Name:     PID_LOGIC - PID_LOGIC_Struct
-- Project Name:    MB_PID2.ise
-- Target Devices:  ML310 development board
-- Tool versions:   8.2i
--
-- Description:     This module is a behavioral definition of the operation of the
--                  logic that captures the
--                  the PID from the PPU.  When the PID is captured it is compared to
--                  see if it matches one of the "stored" PIDs.  If a match occurs,
--                  then the corresponding interrupt signal is triggered.  This module
--                  only can output two different interrupts, so only two different
--                  contexts can be monitored.  This is not only a limitation of how we
--                  have coded this module, but also a limitation on we have
--                  implemented the embedded system in EDK (i.e. we designed the
--                  interrupt controller to allow only 2 interrupts.  If the PID does
--                  not match one of the "stored" PIDs, then no interrupt is
--                  triggered.  This logic also stores the current PID, so subsequent
--                  accesses to other memory addresses will not affect the captured
--                  PID of the of the "currently executing process".  The PIDs that
--                  trigger the interrupts can be changed by changing the "pid1" and
--                  "pid2" variables.
--
-- Dependencies:    This file is required by MB_PID2_top.vhd module in the
--                  MB_PID2.ise project
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

```

```

entity PID_LOGIC is
    PORT(
        ADDR_IN : IN std_logic_vector(0 to 31); --OPB address lines
        DATA_IN : IN std_logic_vector(0 to 31); --OPB data lines
        RNW_IN   : IN std_logic;                --OPB read/write signal
        CLK_IN   : IN std_logic;                --system clock
        RST_IN   : IN std_logic;                --system reset
        INT_A_OUT : OUT std_logic;               --PID A interrupt
        INT_B_OUT : OUT std_logic;               --PID B interrupt
    );
end PID_LOGIC;

architecture PID_LOGIC_Behavior of PID_LOGIC is

    SIGNAL DATA_VAL : std_logic_vector(31 downto 0);
    SIGNAL COMP_VAL  : std_logic_vector( 1 downto 0);

begin

    Capture : Process(RST_IN, CLK_IN, RNW_IN)

        --address where the PID resides in memory
        VARIABLE pid_addr : std_logic_vector(31 downto 0) := X"30000001";

        --temp storage for the PID on the OPB data bus
        VARIABLE pid_val  : std_logic_vector(31 downto 0) := X"00000000";

        --PID of the first process to be monitored
        VARIABLE pid1     : std_logic_vector(31 downto 0) := X"30001000";

        --PID of the second process to be monitored
        VARIABLE pid2     : std_logic_vector(31 downto 0) := X"30002000";

    Begin
        IF (RST_IN = '0')
            THEN
                pid_VAL := X"00000000";
                INT_A_OUT <= '0';
                INT_B_OUT <= '0';
            ELSIF (CLK_IN = '1' AND CLK_IN'LAST_VALUE = '0' AND RNW_IN = '0' AND
                    ADDR_IN = pid_addr)

```

```

        THEN
            pid_val := DATA_IN; --captures PID if a PID is written to memory
        END IF;

--interrupt control logic
IF (pid_val = pid1) then    --triggers interrupt A if PID matches pid1
    INT_A_OUT <= '1';
    INT_B_OUT <= '0';
ELSIF (pid_val = pid2) then --triggers interrupt B if PID matches pid2
    INT_A_OUT <= '0';
    INT_B_OUT <= '1';
ELSE
    INT_A_OUT <= '0';    --no interrupts triggered if PID does not match
    INT_B_OUT <= '0';    --pid1 or pid2
END IF;
End process Capture;

end PID_LOGIC_Behavior;

```

Appendix B. Development Software Tutorials

B.1 Embedded Linux Tutorial

This section presents a guide for creating a relatively simple embedded design using the XUPV2P development board and the Xilinx development environment (version 8.1i). We will discuss preparing the the environment, creating a reference system using the Xilinx Base System Builder, using Platform Studio to modify the design, and using a CF card to load code when turning the system on. The goal of this section is to provide the basic knowledge needed to create, modify, and implement an embedded design quickly in order to familiarize the reader with the development environment. If any problems are encountered that the Tutorial or Troubleshooting sections do not cover, please refer to [63,64].

B.1.1 Initializing The Environment. The first thing that must be done is to make sure that all the required software tools are present. Both Xilinx ISE Foundation 8.1i and EDK 8.1i must be installed on a Windows XP-based machine. Although it is probably not necessary, it is also a good idea to download the most recent IPCores updates and service packs. These can be found at the Xilinx website. Finally, the design repository that comes with the XUPV2P board must be copied to any location on the harddrive. The repository files are responsible for making the Xilinx software aware of all of the features of the XUPV2P development board so

that they can be configured. The repository directory can be found on the CD that came with the XUPV2P.

B.1.2 The Base System Builder. The Base System Builder, known as the BSB hereafter, is a design tool that is part of the EDK Platform Studio which provides a quick, semi-automated method for creating and implementing embedded designs. The BSB has certain limitations, however, that prevent it from being able to create multi-processor designs and adding additional IPCores to a design. This limitation will be addressed in the next section. Despite such limitations, the BSB provides a convenient and efficient means to build a working reference embedded design that can later be built upon.

In order to access the BSB, you must run the EDK Platform Studio. This can be found under the EDK directory of the Windows “Start” menu. Upon starting Platform Studio, you will be greeted with a prompt asking what you would like to do. You can use the BSB, create a blank project, or open an already existing project. Select “Base System Builder wizard” and click “OK”. A prompt will be displayed asking where you would like to save the EDK project file and if you would like to include a design repository. Choose a desired path and project name to store your project information, making sure there are no spaces in the path name as this can cause problems later when trying to implement your design. Also select the “Use Repository Paths” checkbox and point the BSB to the “lib” folder within the directory you copied the XUPV2P repository files to earlier. The next window asks if

you would like to create a new design or open an exiting .bsb design. Select the create new design option. A menu will then open asking what the target development board is. Use the dropdown menus to choose the Xilinx XUP Virtex-II Pro Development System Revision C board and click “Next”.

The next several menus are used to choose what components of the development board you would like to implement in your design. Select the check boxes for the following components and use defaults unless otherwise stated:

- 1. PowerPC Core (w/ Cache Setup Enabled)
- 2. RS232_Uart_1
- 3. SysACE_CompactFlash_1 (use interrupt)
- 3. LEDs_4Bit
- 4 DIPSWs_4Bit
- 5. PushButtons_5Bit
- 6. DDR RAM corresponding to your hardware configuration
- 7. plb_bram_if_cntlr_1 (128KB)

Once the hardware components have been chosen, you can choose 2 tests to include in your design: 1) a memory test and 2) a peripheral selftest. Select both tests as you will be able to choose which test to implement later. Also select RS232_Uart_1 from both the STDIN and STDOUT dropdown menus and then click “Next”. The next prompt will allow you to configure what memory location on the development

board your programs will be run from. For the memory test, select the `plb_bramif_cntlr_1` for all dropdown menus. For the peripheral test, select `DDR_SDRAM_1`. if you have installed RAM, or `plb_bram_if_cntlr_1` if you are not using RAM. Click “Next”.

A summary of the components and their corresponding addresses you have added o your design will be displayed. Look this over and make sure you made no mistakes, selecting “Back” if you have to change anything. When done, select “Generate”. Now is a tricky part. If you select the Platform Studio window and look at the Console Window at the bottom of the screen, you will see numerous “Unknown DIR value UNKNOWN” errors in the *.mhs file. This must be fixed before you select ”Finish”. In order to do this, navigate to the directory where you saved the project file to earlier. This directory now contains number of new directories and configuration files to tell the EDK how to create the bitstream that will be used to configure the FPGA based on the components you selected earlier. Open the system.mhs file using a simple text editor like notepad. Towards the top of the .mhs file there will be 7 “PORT” variables with the “DIR” attribute initialized to “UNKNOWN”. Comment out these lines of code by placing a “#” at the beginning of each line and save the file and exit. Now go back to the BSB wizard and select “Finish”. The required files will then be checked for errors and a new menu will open giving you a choice of what to do next. Make sure that the development board is powered on and the the USB cable connecting the board to the windows box is connected. Select the “Download the design to the board and test it” option. This will synthesize the design, build the

memory test, and then download them to the board. This may take anywhere from 5 to 10 minutes depending on your system.

Now that the design has been created and downloaded to the board it is time to see if it works. Use a serial cable to connect the board's serial port to the machine with Platforma Studio on it. Open a hyperterminal window and configure it to 9600 baud, 8 data bits, no parity, 1 stop bit, no flow control, and to the serial ports COM port. Press the restart button on the board. If all goes well you should see something in the hyperterminal window. If using RAM, you should see a number of tests. If the any result of these test failed, you either selected the wrong RAM option when configuring your design or you need to try using different RAM. If you are not using RAM, and hence are using `plb_bram_if_cntlr_1`, all you should see is "entering main" followed by "exiting main". This is because you cannot test the memory where the memory test is actually residing.

B.1.3 Platform Studio. Platform Studio is the heart of the development environment. Once the BSB has been completed, the Platform Studio interface will be updated to reflect your project. On the left side of Platform studio are three tabs: Project, Applications, and IP Catalog. The project tab lists the project files, general options, and reference files. These should not need to be altered in any way. The applications tab displays which programs are associated with your project and allows you to select which program to load to the board, view source code, and modify program attributes like the linker script which will be important later. The IP catalog

conveniently lists all of the IP cores that can be loaded into a design and allows you to do so.

The right window of Platform Studio is your main view window. This is where your system assembly and overall block diagram of your design can be viewed. The system assembly tab provides information on how all of the IPcores are linked together to form your design. You can view this from the bus interface, port, or memory address perspectives by selecting the appropriate radio button at the top of the tab. You can modify names, links, and hardware addresses in these perspectives as well. The block diagram will not be displayed until your design has been compiled. The block diagram just serves as a convenient means to see your design's configuration. You should take a moment to explore the Platform Studio GUI in order to familiarize yourself with it.

In order to run the peripheral selftest, it must be selected as the application to run and be compiled. To do this, click on the "Applications" tab. Right click on the TestApp_Memory project and uncheck "Mark to initial BRAMs". Next, right click on the TestApp_Peripheral and select "Mark to initialize BRAMs". This will set the peripheral selftest as the project to download to the board. If using RAM, right click on "TestApp_Peripheral" and select "Build Project". If using plb_bram_if_cntlr_1 then right click "TestApp_Peripheral" and select "Set Compiler Options". Under the "Environment" tab in the window that pops up, select the "Use default Linker Script" checkbox. Set the Program start address to "0x00000200", and the stack and the heap sizes both to "400" and click "OK". Now right click the peripheral selftest project and select "Build Project". Check the console window to ensure that there were no errors

during build. Now select the “Device Configuration” menu at the top of the Platform Studio Window and choose “Download Bitstream”. When it is done downloading, go back to your hyperterminal and press the reset button on the development board. You should see a tests for the SysACE, LEDS, etc. All test should pass except for the SysACE since a CF card is not plugged in. Also, you can change the dipswitch configuration and hold the push-buttons on the development board while resetting the development board to see different values returned during the selftest.

B.1.4 Compact Flash and SysACE. While a compact flash card is not required to get an embedded system up and running, using one has many advantages but doing so is not necessarily straightforward. First, the XUPV2P board is very particular about the file system of the CF card, thus it must be properly formatted. In order to do this, a Windows version of mkdosfs.exe is needed. This can be found easily by doing an internet search. After downloading mkdosfs.exe, place it in an easily accessible directory. Make sure the CF card is attached to the machine using a CF card reader. Open a dos command prompt and navigate to directory containing mkdosfs.exe and type the following command:

“mkdosfs -s 64 -F 16 -R 1 X:”

where X: is the drive letter of the compact flash card. This will format the CF card using a FAT16 filesystem with 64 sectors per cluster and 1 reserved sector.

Now that the CF card is ready, the SysACE file that will go on the CF card is to be generated. First the GenACE.opt file must be created. Create a new document in notepad and enter the following:

```
-jprog  
-board user  
-target ppc_hw  
-hw implementation/download.bit  
-elf TestApp_Peripheral/executable.elf  
-configdevice devicenr 1 idcode0x1127e093 irlength 14 partname xc2vp30  
-debugdevice devicenr 1cpunr 1  
-ace system.ace
```

Save the file as “GenACE.opt” in the project directory. Next, open a cygwin shell by selecting the “Launch EDK shell:.” from the “Project” dropdown menu in Platform studio. In the shell, navigate to your project directory and type the following command:

```
“xmd -tcl genace.tcl -opt GenACE.opt”
```

This will create a file named system.ace in the project directory. Copy this file to the CF card that was formatted earlier and insert it into the CF card slot on the development board (make sure the power is off). Now turn the board on. If all has gone correctly, the ACE LED on the board should be solid green, rather than blinking red. Also, the peripheral selftest should have executed with similar results to when it was run previously. The only difference should be that the SysACE test will now be passed.

B.2 Troubleshooting

In this section we cover the problems that were encountered over the course of our development thus far. Although this is not meant to be all inclusive, it is a convenient place to begin to find a solution, and it may shed some light on a similar problem you may be encountering.

B.2.1 Development Environment.

- Q1. Why does the BSB return an error when I try to tell it where to store the project?
- A1. While the BSB asks for a directory, you must also provide a name for your EDK project. Also, remember to make sure that the filename and path do not have any spaces in them.
- Q2. I just recompiled my software and now it freezes in the middle of execution. What should I do?
- A2. This can especially be if a problem if you chose to store your program in BRAM. Make sure that your linker is set to use the "default linker script" and that the program start address is above 0x00000100 to avoid overriding the interrupt vector jump table. Also, since BRAM is at 128KB at its maximum, its addresses ranges from 0x00000000 to 0x00020000. Thus make certain that the size of your stack and heap are small enough to not exceed the 128KB limit after you account for your program size. Note: Depending on how you configured

your board, you may have less than 128KB of BRAM, so keep track of your memory address range accordingly.

- Q3. Why can't I generate a system.ace file? It keeps exiting on an error. /item
A3. make sure that there are no spaces in any of your paths to files that may be being used by the system ace utility. This includes the "My Documents" folder, so you can not use that to store your project information.
- Q4. The BSB returns an error when I try to "Finish" the wizard. What can I do?
- A4. For some reason, when using the XUPV2P development board and the BSB, it will try to initialize some variables within the project's .mhs file that are not there. This causes the .mhs file to cause the final build script to crash. Thus, before clicking "Finish" in the BSB, open the .mhs file in your projects directory, and comment out the lines causing the problem. To figure out what lines you need to comment, check the Console Window of the Platform Studio. This window should report what lines are causing the problem.

B.2.2 General Linux.

- Q1. When running certain commands in linux, I get a permission denied message. What should I do?

- A1. This error is due to your permissions in linux. There are a number of commands that can change either the folder permissions - such as `chmod`, `chgrp`, and `chown` - or your user's permission level - such as `gpasswd`. Information on all such commands can easily be found on the internet. Experiment with these commands to learn them as they will all be very handy. Alternatively, you can use the “su-” command to switch to root privileges if you have root access, but this can be dangerous. You must be a member of group “wheel”s to be able to do this.

B.2.3 Embedded Linux Installation.

- Q1. While creating the crosscompiler the command shell returns an error when using the “setenv” command.
- A1. This depends on the shell you are using. Replace “setenv” with “export”.
- Q2. When trying to download the Linux sources using bitmover, I get an error saying that it cannot find `sfio.sh`. I see a `sfio.sh` in the bitmover directory, however. What should I do?
- A2. you need to temporarily add your bitmover directory to your execution path. This can be done by typing, “`PATH=|bitmover directory path|:$PATH`”, in the console.

- Q3. When I try to run the `mkrootfs.sh` script, all the directories are created, but no system programs have been installed. Why is busy box not working?
- A3. The problem is not with busybox itself; it is with the script. The `mkrootfs.sh` script has a busy box directory variable about 80to where you have installed busybox to.
- Q4. I just formatted and partitioned my CF card and put the `.ace` file and root file system on it. Now linux won't boot up at all at system power-on anymore.
- A4. As the XUPV2P is particular about how the CF card is partitioned, the `fdisk` formatter messed it when you made the boot partition. To fix this, first, with all three partitions still on the CF card, use the process from the Tutorial section to format the card. Then go back into `fdisk` on the linux box and create the swap and root partitions again. Make sure not to use `fdisk` to make the boot partition. Do, however, make the boot partition as bootable. Now copy the root filesystem back onto the root partition. Place the CF card in the CF card slot and power the system on.
- Q5. Linux starts and a login prompt appears. When I try to login as root, it returns me back to another login prompt.

- A5. Make sure that the force dedicated serial console in the busybox config is not selected. Also, make sure that you have chosen a shell (the ash shell is very close to the bash shell) in the busybox config.

Bibliography

1. Anderson, Debra, Thane Frivold, and Alfonso Valdes. *Next-generation Intrusion Detection Expert System (NIDES): A Summary*. Technical report, Computer Science Laboratory, 1995.
2. Arbaugh, William H., David J. Farber, and Jonathan M. Smith. "A Secure and Reliable Bootstrap Architecture". *1997 IEEE Security and Privacy Conference*. 1997.
3. Arora, Divya, Srivaths Ravi, Anand Raghunathan, and Niraj Jha. "Secure Embedded Processing Through Hardware-assisted Run-time Monitoring". *Design, Automation and Test in Europe Conference and Exhibition*. 2005.
4. Axelsson, Stefan. *Intrusion Detection Systems: A Survey and Taxonomy*. Technical report, Chalmers University of Technology, March 2000.
5. Bajikar, Sunjeep. *Trusted Platform Module (TPM) based Security on Notebook PCs - White Paper*. Technical report, Intel Corp., June 2002.
6. Baker, Zachary K. and Viktor K. Prasanna. "Highthroughput LinkedPattern Matching for Intrusion Detection Systems". *ANCS'05*. October 2005.
7. Brumley, David. "invisible intruders: rootkits in practice". *login: The USENIX Magazine*, 1999. <http://www.usenix.org/publications/login/1999-9/features/rootkits.html>.
8. Burke, Dan, James Player, and Nacho Navarro. "Building a Xilinx ML300/ML310 Linux Kernel". November 2004. <http://www.crhc.uiuc.edu/IMPACT/gsrc/hardwarelab/docs/kernel-HOWTO.html>.
9. CACI. "Computer Security Threats: Malicious Threats". 2007. <http://www.caci.com/business/ia/threats.html>.
10. CERT, (Computer Emergency Response Team). "CERT/CC Statistics 1988-2006". 2006. www.cert.org/stats/cert_stats.html.
11. Chari, Suresh N. and Pau-Chen Cheng. "BlueBoX: A Policy-Driven, Host-Based Intrusion Detection System". *ACM Transactions on Information and System Security*, volume 8, 173–200. May 2003.
12. Cho, Young H. and William H. Mangione-Smith. "A Pattern Matching Coprocessor for Network Security". *DAC 2005*. June 2005.
13. Clark, Chris, Wenke Lee, David Schimmel, Didier Contis, Mohamed Kon, and Ashley Thomas. "A Hardware Platform for Network Intrusion Detection and Prevention". *Third Workshop on Network Processors & Applications (NP3)*. February 2004.

14. Coburn, Joel, Srivaths Ravi, Anand Raghunathan, and Srimat Chakradhar. "SECA: Security-Enhanced Communication Architecture". *CASES'05*. September 2005.
15. F-Secure. "F-Secure Virus Descriptions : Slammer". 2007. <http://www.f-secure.com/v-descs/mssqlm.shtml>.
16. Feng, Henry Hanping, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. "Anomaly Detection Using Call Stack Information". *Inproceedings of the 2003 IEEE Symposium on Security and Privacy*. 2003.
17. Garfinkel, Tal and Mendel Rosenblum. "A Virtual Machine Introspection Based Architecture for Intrusion Detection". *Proceedings of the 2003 Network and Distributed System*. 2003.
18. Goldberg, Ian, David Wagner, Randi Thomas, , and Eric Brewer. "A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker)". *Proceedings of the Sixth USENIX UNIX Security Symposium*. 1996.
19. Grevstad, Eric. "CPU-Based Security: The NX Bit". May 2004. <http://hardware.earthweb.com/chips/article.php/3358421>.
20. Gupta, R. Sekar and A., J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. "Specification-based Anomaly Detection: A New Approach for Detecting Network Intrusions". *CCS'02*. Nov 2002.
21. Held, Jim, Jerry Bautista, and Sean Koehl. *From a Few Cores to Many: A Tera-scale Computing Research Overview*. Technical report, Intel Corp., February 2007.
22. Huang, Yian and Wenke Lee. "A Cooperative Intrusion Detection System for Ad Hoc Networks". *1st ACM Workshop Security of Ad Hoc and Sensor Networks*. 2003.
23. IBM. *How to Secure an Insecure OS*. Technical report, IBM Corp., 2002.
24. Intel. *LaGrande Technology Architectural Overview*. Technology Overview 252491-001, Intel Corp., September 2003.
25. Intel. "System Programming Guide, Part 1". *Intel 64 and IA-32 Architectures Software Developers Manual*, 3A, 2006.
26. Intel. "System Programming Guide, Part 2". *Intel 64 and IA-32 Architectures Software Developers Manual*, 3B, 2006.
27. Kim, Gene and Eugene H. Spafford. "The Design and Implementation of Tripwire: A File System Integrity Checker". *Proceedings of the 2nd ACM Conference on Computer and Communications Security*. 1994.
28. King, Samuel T., Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. "SubVirt: Implementing Malware With Virtual Machines". *Proceeding of the 2006 IEEE Symposium on Security and Privacy*. 2006.

29. Kiriansky, Vladimir L. *Secure Execution Environment via Program Shepherd*. Master's thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, February 2003.
30. Klingauf, Wolfgang and Uwe Klingauf. "Virtex2Pro & Linux". January 2004. <http://www.klingauf.de/v2p/index.phtml>.
31. Ko, Calvin Cheuk Wang. *Execution Monitoring of Security-critical Programs in a Distributed System: A Specification-based Approach*. Ph.D. thesis, University of California - Davis, 1996.
32. Kumar, Sandeep and Eugene H. Spafford. "A Software Architecture to Support Misuse Intrusion Detection". In *Proceedings of the 18th National Information Security Conference*, 194–204. 1995.
33. Kuperman, Benjamin A. *A Categorization of Computer Security Monitoring Systems and the Impact on the Design of Audit Sources*. Ph.D. thesis, Purdue University, 2004.
34. Lee, Ruby B., David K. Karig, John P. McGreggor, and Zhijie Shi. *Enlisting Hardware Architecture to Thwart Malicious Code Injection*. Technical report, Princeton University, 2003.
35. Li, Shaomeng, Jim Torresen, and Oddvar Soraasen. "Exploiting Reconfigurable Hardware for Network Security". *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 292–293. 2003.
36. Lipton, R.J., S. Rajagopalan, , and D.N. Serpanos. "Spy: A Method to Secure Clients for Network Services". *22nd International Conference on Distributed Computing Systems Workshops*. 2002.
37. Litty, Lionel. *Hypervisor-Based Intrusion Detection*. Master's thesis, University of Toronto, 2005.
38. Manadhata, Pratyusa and Jeannette Wing. *An Attack Surface Metric*. Technical Report CMU-CS-05-155, Carnegie Mellon University, 2005.
39. Messerges, Thomas S. and Ezzat A. Dabbish. "Digital Rights Management in a 3G Mobile Phone and Beyond". *DRM'03*. October 2003.
40. Molina, Jesus and William Arbaugh. "Using Independent Auditors as Intrusion Detection Systems". *Information and Communications Security: 4th International Conference*. December 2003.
41. Nagarajan, Raj and Vasanth Asokan. *Getting Started with uClinux on the MicroBlaze Processor*. Technical Report XAPP730, Xilinx, Inc., September 2006.
42. Nelson, Brent and Brad Baillio. "Configuring and Installing Linux on Xilinx FPGA Boards". November 2005. <http://splish.ee.byu.edu/projects/LinuxFPGA/configuring.htm>.

43. Ng, Harn Hua. *PPC405 Lockstep System on ML310*. Technical Report XAPP564, Xilinx, Inc., 2004.
44. Nguyen, Tuan. "AMD Announces "Torrenza" Technology". *Daily Tech: IT*, June 2006. <http://www.dailytech.com/article.aspx?newsid=2642>.
45. One, Aleph. "Smashing The Stack For Fun And Profit". *Phrack*, Volume 7, Issue 49, November 1996. <http://www.cs.wright.edu/people/faculty/tkprasad/courses/cs781/alephOne.html>.
46. Petroni, Nick L., Timothy Fraser, Jesus Molina, and William A. Arbaugh. "Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor". *Proceedings of the 13th USENIX Security Symposium*. USENIX, August 2004.
47. Ptacek, Thomas H. and Timothy N. Newsham. *Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection*. Technical report, Secure Networks, Inc., 1998.
48. Ragel, Roshan G., Sri Parameswaran, and Sayed Mohammad Kia. "Micro Embedded Monitoring for Security in Application Specific Instruction-set Processors". *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 304–314. 2005.
49. Saggese, G. P., C. Basile, L. Romano, Z. Kalbarczyk, and R. K. Iyer. "Hardware Support for High Performance, Intrusion- and Fault-Tolerant Systems". *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*. 2004.
50. Shi, Weidong, Hsien-Hsin S. Lee, Guofei Gu, Laura Falk, Trevor N. Mudge, and Mrinmoy Ghosh. "An Intrusion-Tolerant and Self-Recoverable Network Service Using A Security Enhanced Chip Multiprocessor". *ICAC'05*. 2005.
51. Silberschatz, Abraham, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., 2005.
52. Smaha, S.E. "Haystack: An Intrusion Detection System". *Aerospace Computer Security Applications Conference, Fourth*. 1988.
53. Song, Hayou and John W. Lockwood. "Efficient Packet Classification for Network Intrusion Detection using FPGA". *FPGA '05*. February 2005.
54. Sun. "UltrarSparc T1 Specifications". 2006. <http://www.sun.com/processors/UltraSPARC-T1/specs.xml>.
55. Sundaramoorthy, Navaneethan, Raj Nagarajan, and Vasanth Asokan. *A Portable uClinux Development Environment on a Windows PC*. Technical Report XAPP934, Xilinx, Inc., September 2006.
56. Wikipedia. "NX bit". http://en.wikipedia.org/wiki/NX_bit#Hardware_background.
57. Wikipedia. "Rootkit". 2007. <http://en.wikipedia.org/wiki/Rootkit>.

58. Williams, Paul D. *Warthog: Towards a Computer Immune System for Detecting "Low and Slow" Information System Attacks*. Master's thesis, Air Force Institute of Technology, March 2001.
59. Williams, Paul D. *CUPIDS: Increasing Information System Security Through The Use of Dedicated Co-Processing*. Ph.D. thesis, Purdue University, August 2005.
60. Xilinx. *Development System Reference Guide*. Xilinx, Inc., 8.2i edition.
61. Xilinx. *PowerPC Reference Guide*. Xilinx, Inc., 1.1 edition, September 2003.
62. Xilinx. *EDK OS and Libraries Reference Guide (UG114)*. Xilinx, Inc., 3.0 edition, June 2004.
63. Xilinx. *Embedded System Tools Reference Manual (UG111)*. Xilinx, Inc., 5.0 edition, October 2005.
64. Xilinx. *Getting Started with EDK*. Xilinx, Inc., 8.1i edition, November 2005.
65. Xilinx. *ML310 User Guide: Virtex-II Pro Embedded Development Platform (UG068)*. Xilinx Inc., v1.1.4 edition, January 2005.
66. Xilinx. *PowerPC Block Reference Guide (UG018)*. Xilinx, Inc., v2.1 edition, July 2005.
67. Xilinx. *Virtex-II Pro and Virtex-II Pro X FPGA User Guide (UG012)*. Xilinx, Inc., v4.0 edition, March 2005.
68. Xilinx. *MicroBlaze Processor Reference Guide (UG081)*. Xilinx, Inc., 6.0 edition, June 2006.
69. Xilinx. *MPMC2 Release Notes*. Xilinx, August 2006.
70. Xilinx. *Multi-Port Memory Controller 2 GUI User Guide (UG245)*. Xilinx, Inc., v1.0.1 edition, April 2006.
71. Xilinx. *Multi-Port Memory Controller 2 User Guide (UG253)*. Xilinx, Inc., v1.1 edition, August 2006.
72. Zhang, Tao, Xiaotong Zhuang, Santosh Pande, and Wenke Lee. *Hardware Supported Anomaly Detection: Down to the Control Flow Level*. Technical Report GIT-CERCS-04-11, Center for Experimental Research in Computer System at Georgia Institute of Technology, 2004.
73. Zhang, Tao, Xiaotong Zhuang, Santosh Pande, and Wenke Lee. "Anomalous Path Detection with Hardware Support". *CASES'05*. September 2005.
74. Zhang, Xiaolan, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. *Secure Coprocessor-based Intrusion Detection*. Technical report, IBM Corp.

Vita

Second Lieutenant Stephen D. Mott graduated from Palmer High School in Colorado Springs, CO. He entered undergraduate studies at the United States Air Force Academy in Colorado Springs, CO where he graduated with a Bachelor of Science in Electrical Engineering and received a regular commission in June 2005. In August 2005, he entered the Graduate School of Engineering and Management, Air Force Institute of Technology as a direct accession from his undergraduate program. Upon graduation, he will be awarded a Masters of Science in Electrical Engineering and be assigned to the Air Force Research Lab.

REPORT DOCUMENTATION PAGE					<i>Form Approved OMB No. 0704-0188</i>	
<small>The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.</small>						
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.						
1. REPORT DATE (DD-MM-YYYY)		2. REPORT TYPE			3. DATES COVERED (From - To)	
4. TITLE AND SUBTITLE				5a. CONTRACT NUMBER		
				5b. GRANT NUMBER		
				5c. PROGRAM ELEMENT NUMBER		
6. AUTHOR(S)				5d. PROJECT NUMBER		
				5e. TASK NUMBER		
				5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)					8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)					10. SPONSOR/MONITOR'S ACRONYM(S)	
					11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT						
13. SUPPLEMENTARY NOTES						
14. ABSTRACT						
15. SUBJECT TERMS						
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON	
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER (Include area code)	